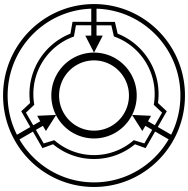
TU WIEN Informatics

VS Code Language Extension - MiniProb



BACHELORARBEIT

zur Erlangung des akademischen Grades

Bachelor of Science

im Rahmen des Studiums

Bachelor programme Software & Information Engineering

eingereicht von

Rafael Tanzer

Matrikelnummer 12224207

an der Fakultät für Informatik
der Technischen Universität Wien

Betreuung: Ezzio Bartocci
Mitwirkung: Michele Chiari
Francesco Pontiggia

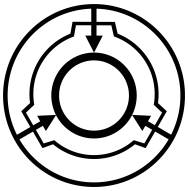
Vienna, 24 May 2025

Rafael Tanzer

Ezzio Bartocci

TU WIEN Informatics

VS Code Language Extension - MiniProb



BACHELOR'S THESIS

for the degree of

Bachelor of Science

within the study programme of

Bachelor programme Software & Information Engineering

submitted by

Rafael Tanzer

Student ID 12224207

to the Faculty of Informatics
at the TU Wien

Advisor:	Ezio Bartocci
Assistance:	Michele Chiari
	Francesco Pontiggia

Vienna, 24 May 2025

Rafael Tanzer	Ezzio Bartocci

Abstract

??all slash code coantined text hs to be revisted due to the detokenize

Cyber-Physical Systems demand rigorous, automated verification to ensure safety and correctness under uncertainty. This thesis presents the design and implementation of a Visual Studio Code language extension for **MiniProb** (4.4), a domain-specific probabilistic programming language tailored to formal model checking of recursive, stochastic processes. Building on the Langium framework - written in TypeScript and running on Node.js - and the Language Server Protocol, a formal EBNF grammar for MiniProb was defined and used within Langium to generate a parser that produces both Concrete Syntax Trees (CSTs) and typed Abstract Syntax Trees (ASTs). Custom services - including semantic validation, type checking, cross-reference resolution, and some code completion features - were developed within Langium.
The extension, packaged as a .vsix and available on GitHub, streamlines MiniProb development by lowering the barrier to entry for probabilistic model authors and facilitating direct translation of MiniProb programs into probabilistic operator precedence automata for formal verification.
This work demonstrates that modern editor tooling, when coupled with robust language engineering frameworks, can significantly enhance the usability and accessibility of specialized DSLs in safety-critical CPS contexts.

Contents

Part I

Foundations

Introduction

1.1 Motivation

The **MiniProb** DSL, created by the supervising team, provides a concise notation for probabilistic process models[12]. At present, MiniProb tooling is limited to a Haskell-based parser, yielding a tedious workflow: source code is written in a generic editor and syntax errors discovered one at a time as the parser aborts on its first failure. A modern editor extension - for example, a Visual Studio Code plugin providing syntax highlighting and live diagnostics - would dramatically streamline development. Such an interface lowers the barrier to entry for new users and also offers a familiar environment in which MiniProb programs can be authored and then further automatically converted into automata for formal verification [29, 30]. In the context of cyber-physical systems - where interconnected devices underpin critical infrastructure - and the ever-growing complexity and threat landscape of cyberspace, integrated verification is essential to ensure system reliability, preempt security vulnerabilities, and uphold correctness guarantees under rigorous specifications.

1.2 Objectives and Scope

The main goal of the thesis was to create a *Language Extension* 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 (2.1), convert written text into abstracted parts (2.2.1).

In this thesis, a generative context-free grammar is used to formally describe *MiniProb* 4.4 and a domain-specific language (DSL) is designed (8), for authoring *POMC* files. With a formal language definition as foundation, parsers enable 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. This includes syntactic and semantic analysis through syntax highlighting and validation, accurate resolution of symbol references and rudimentary code completion, as well as implemented type checking mechanisms to ensure compatibility during development.

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 service as the implementation evolves.

Finally, the complete source code is hosted on GitHub at `https://github.com/e12224207/miniprob`, and prebuilt `.vsix` packages can be downloaded from the Releases page for straightforward installation. This documentation serves as the definitive guide to the application developed in that repository, underpinned by

theoretical foundations to contextualize the themes of this thesis.

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.[20] 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 transformed into a regular grammar, and vice versa. Further, every grammar in the Chomsky hierarchy can be characterized by an equivalent acceptor automaton, which admits all input that ends in valid states as words of the language. 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 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[3], 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 while preserving structural clarity and high readability. This declarative approach not only supports rigorous specification but also facilitates parsing, 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. [22, 21]
Within the scope of this thesis, the Extended Backus-Naur Form (EBNF) is particularly significant, since Langium employs its own custom EBNF dialect (8) to define the grammars underpinning its language-assistance features.

2.2 Language Structure and Paradigms

2.2.1 Parsing Models

The core functionality of language extensions stems from *parsers* which, against a formal language definition, convert written text - most often program code - into abstracted parts. 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. Such specifications range from various types of grammars to automata, and provide the structural and syntactic rules necessary for correct interpretation and validation of the 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 of it occurs.

Parsing strategies play a central role in the implementation of programming languages, determining how source code is analyzed and transformed into syntax trees. Two primary streams dominate the landscape: top-down and bottom-up parsing. Each parsing approach encompasses distinct implementation strategies and imposes specific constraints on grammar design, thereby influencing the overall complexity of language implementation.

Top-down parsers, such as those following the *LL(k)* model, process input from left to right, constructing parse trees from the root downward. These parsers require grammars to avoid *left recursion*, a situation where a non-terminal refers to itself as the first symbol on the right-hand side of a production (e.g., $\mathtt{A} \rightarrow \mathtt{A}\ \alpha$). Left-recursive rules can lead to infinite recursion during top-down parsing. In contrast, *bottom-up* parsers, including *LR(k)* and its variants still read left to right but build parse trees from leaves to root and can naturally accommodate left-recursive structures.

The parameter k in LL(k) and LR(k) denotes the number of lookahead tokens needed to unambiguously select production rules. Larger values of k allow more expressive grammars but may increase the complexity of the parser and slow down parsing. As such, grammars used in practice are often simplified or transformed to be compatible with the limitations of the chosen parsing model.

Understanding these parsing models is crucial when designing or analyzing a programming language, especially when employing parser generators such as ANTLR and Chevrotain (LL) or Yacc/Bison (LR), which impose specific requirements on grammar structure.

2.2.2 Abstract vs. Concrete Syntax

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. [32] 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. [1]

2.2.3 Imperative vs. Declarative Languages

Languages used for further computation can be analyzed from multiple perspectives and classified with different qualities in mind. [34] For this thesis we focus on slotting languages into two overarching paradigms, **imperative** and **declarative** languages, which differ primarily in how they express the path toward a desired outcome. Declarative languages describe *what* the computation should accomplish by adhering to sets of expressions, constraints or logical statements. [35] 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.

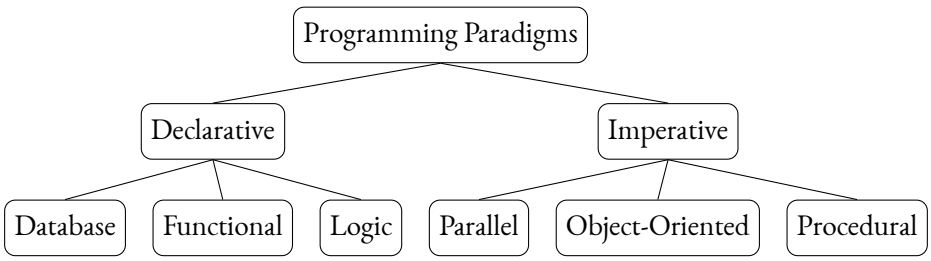


Figure 2.1: Commonly encountered programming paradigms.

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 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.

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 themselves. The imperative paradigm reflects von Neuman machines: 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 shifted in and out of focus depending on the context of the computation. Such techniques encompass memory and cache management, *Instruction-Set* optimization or compiler optimization.

However, while this explicit control provides developers with great freedom, it also places a greater burden on them to manage every detail of the 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. [27] 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 include 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. DSL 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 a 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 [14], 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 the 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 applicable imperative DSL is often simpler than creating a declarative one: an interpreter processes each command in sequence, or a translator converts DSL instructions into a general-purpose language.

Imperative DSLs are well suited for scripting and defining macros, where users sequence domain-specific commands to drive behavior. They also shine in combination with 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 to program custom action sequences - such as picking up an object, then placing it - 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 benefits from the clarity and control that an imperative DSL provides.

Cyber-Physical Systems and Formal Analysis

3.1 Cyber-Physical Systems: Definition and Examples

Cyber-Physical System (CPS) is an umbrella term broadly referring to any systems which tightly integrate both computational logic (cyber) and physical components, allowing interplay through designated interfaces. [24] The two parts of a CPS have to be seamlessly interconnected enabling efficient communication and consistent information exchange [25], as both sides heavily depend on one another. The cyber-physical relationship can be generally describes by three roles *controller/agent*, *sensors* and *activators*, which are connected by an underlying network. Controllers, representing the 'cyber', are occupied with computation and relying fitting instructions based on the observational data provided by the physical sensors. Sensor and Actuators embody the 'physical' and are the ones interacting with the environments in which the system resides. A sensors jobs is to gather intel from its surroundings communicating it in order for the agent to make correct decisions and predictions and to monitor the system. Without sufficient data, the controller is left blind and possibly unable to perform as intended. While sensors passively interact with the environment by observing, actuators do so actively by altering the systems state - carrying out tasks instructed by the controller. This sense->compute->actuate cycle is executed iteratively, forming a continuous feedback loop that enables the system to respond dynamically to changes in its physical environment.

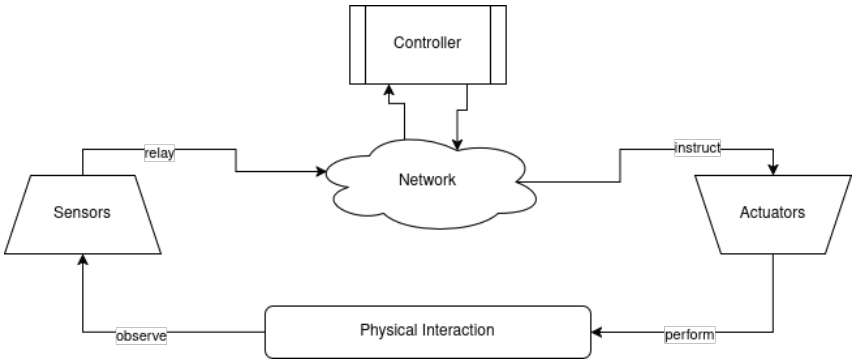


Figure 3.1: General architecture of common CPS

Controllers may range from simple reactive routines to advanced inference models grounded in probabilistic programming, or even intelligent agents capable of autonomous decision-making. Actuators, in turn, translating these decisions into physical action can be represented by basic devices such as motors that move parts, valves that regulate flow, or relays that control circuits. They can also act as more complex assemblies - robotic joints, programmable hydraulic systems, or smart actuators that adjust based on local conditions. At the foundation, sensors continuously monitor the environment, capturing data through modalities such as motion, orientation, or spatial mapping. Together, these elements form an integrated feedback system, enabling CPS to respond in-

telligently and flexibly to their physical context, regardless of application domain.

CPSs are inherently multidisciplinary structures intersecting engineering domains (mechanical, electrical, civil, etc.) with computer science. Because these domains employ differing models for various tasks, thereby effectively combining them is non-trivial. Challenges arise during development and operation of CPSs for which finding solutions is imperative. [25] First and foremost is the concern of **reliability and safety**. Occurrences of failures can have dire real-world consequences with catastrophic end results. This means systems must handle unexpected conditions gracefully and robustly. But ensuring safety through extensive verification and validation is complicated by the combination of continuous physical dynamics and discrete decision logics. Another challenge is **real-time performance**, as computing elements must keep pace with the physical processes, which usually require low-latency responsiveness including the network. Since CPSs are used in infrastructure and other critical sites that may be targets of cyber attacks, guarantying a high standard of **Security** is essential. Additionally, sensors are not completely reliable - possibly introducing noise to measurements - and environmental conditions can vary drastically leading to **uncertainty**, which a CPS has to cope with to be resilient. [33]

3.2 Formal Methods and Model Checking

CPSs are often deployed in safety-critical environments where utmost importance is placed on reliable operation and functional correctness. Yet reaching high levels of confidence in a systems behavior remains extremely challenging inside cyber-physical domains.[2] This difficulty arises from the complex interactions of all the individual integral components of the system, which can not be effectively tested only employing traditional methods. The inherent entanglement of these systems warrants the use of specialized procedures that are able to cover and validate the acting components. Unlike conventional simulation based testing, *formal methods* are able to prove correctness against formal specifications, thereby providing more reliable results and eliminating potential error sources, such as race conditions or violations of safety properties(unnoticed states).

In formal methods, *theorem proving*, *model checking*, and *runtime verification* are widely recognized as the three main approaches for assuring system correctness[23, 2]. All three practices are distinguishable from one another and are tailored for certain scenarios. Runtime verification is a lightweight technique that verifies systems by inspecting their execution trace - often employing automata to detect violations. Theorem proving, in contrast, is an offline and often time intensive process relying on mathematical inference rules to statically validate correctness with a high degree of rigor.

Model checking is of most interest in this case, as the MiniProb DSL 4.4 was developed with this context in mind. [30] It statically verifies a system by exhaustively exploring the state-space and is considered one of the most successful verification techniques. [2] The thorough analysis of reachable states, combined with specifications which are expressible in various forms of modal logic, offer a strong expressive foundation for property verification [4]. However, the systematic exploration of states leads to poor performance, especially for models with complex and big state spaces. Moreover, as most model checking implementation employ automata-based solutions, systems are usually converted into equivalent state machines which can result in an explosion of states.[<empty citation>] Model checking is generally only used in conjunction with finite-state machines for aforementioned reasons. Still, theres been advances trying to apply it to infinite state spaces. [13, 8]. The resulting verdicts for infinite-state models are typically approximations or are restricted by bounded analysis, such as checking only a finite number of steps or abstracting unbounded data domains. As a result, the verification outcomes may not provide absolute guarantees for all possible executions, but rather assurances that hold under specific assumptions, abstractions, or within predefined limits of a system's behavior.

These approaches often rely on *symbolic evaluation* rather than naive enumeration of states. By representing potentially infinite sets of states using logical formulas or data structures such as Binary Decision Diagrams (BDDs) or SMT formulas, symbolic model checking can reason about system behavior without explicitly generating every state.

3.3 Motivation for PPLs in CPS Contexts

Probabilistic programming (3.3) has multiple fitting applications within CPSs, especially where reasoning under uncertainty is essential. Generally, probabilistic programming allows controllers to plan even in unpredictable environments and settings.

- When measurements contain noise or are incorrect, PPL models can still be applied to extract useful information. Instead of treating the raw reading of a sensor as the "truth," the readings are considered samples for a *sensor's probabilistic model*. Not only is it possible to reduce the noise in the measurements by calculating the posterior, but those noise characteristics are also used to flesh out the predictive model itself. This enables PPLs to adaptively adjust noise models online when, for example, sensors deteriorate over time. Additionally - with most CPSs employing multiple sensors - a PPL naturally supports *sensor fusion*, combining data from heterogeneous sources to produce a coherent estimate. By defining each sensor's observation as a conditional distribution, a probabilistic program weights each reading according to its uncertainty: more reliable sensors contribute more heavily, while inconsistent or faulty readings are down-weighted or flagged. Inference then yields a joint posterior over the "true" state, allowing the system to reconcile disagreements between sensors, detect potential sensor failures by inferring latent bias variables, and maintain a quantified confidence.

- A PPL expresses system dynamics probabilistically so that sampling yields a distribution of possible futures [19]. By enforcing "chance constraints" (e.g., "limit the probability of exceeding a safety threshold to 1%"), inference or Monte Carlo sampling identifies control inputs that optimize expected performance while respecting risk. When the current state is uncertain, planning occurs in belief space - each action triggers a Bayesian update of the state distribution, and planners choose sequences that minimize expected cost given that uncertainty. Simultaneously, online Bayesian learning refines model parameters (like friction or drag) over time, tightening future predictions; as uncertainty shrinks, control can push closer to limits, whereas rising uncertainty leads to more conservative actions.

- Which leads us into robotics, a quintessential cornerstone of CPS. Instead of working on raw observational data and making assumptions in incomplete cases, robots can adopt a *belief* based on an observational model [33]. The observational model couples motion dynamics (how actions translate into state transitions) with existing sensor models, providing a full posterior distribution that the robot can use to imply different levels of confidence for its actions. Furthermore, robots often operate in changing environments or with components that degrade over time. Within a PPL, unknown parameters, such as friction coefficients or sensor calibration offsets, can be treated as latent variables and used to perform Bayesian learning to infer them from incoming data. The same inference mechanism used for state estimation thus also adapts the model online, continuously refining both motion dynamics and sensor mappings.

- Recent research is looking at ways to utilize PPLs to gather diagnostics of a CPS. [28] They authors propose a way to infer potential error causes, that might be overlooked otherwise. This of of great interest to system operators for deciding if either parts have to be replaced or only have to be recelebrated. Because measured data on system deviations often reflects complex intertwining and *closed-loop corrections*, identifying root causes becomes a matter of inference. The proposed two-step approach first builds a generative model (based on control software and expert insights) that simulates observations from hypothesized error causes. This simulator is then recast as a probabilistic program, allowing Bayesian inference to estimate latent causes (with confidence intervals) from actual measurements.

Probabilistic Programming

4.1 What Is Probabilistic Programming?

Probabilistic programming is a paradigm that aims to perform statistical analysis using tools yielded by computer science.[26] Unifying general-purpose programming and probabilistic modelling enables the presentation of statistical models as a program. Defined by underlying program code, the model is embodied by the relations between variables and calculations. The established capabilities of programming languages are often augmented with probabilistic functionality, in turn creating probabilistic programming languages (PPL),[17, 29] that are closely associated with this paradigm. The enriched syntax enables the creation of more compact models and fosters conciser abstractions of the underlying probabilistic structure, usually providing constructs that allow declaring random variables and conditioning of observed data. These primitives form the foundation for specifying core elements of statistical models.

Priors are pre-known and usually drawn from distributions reflecting the initial belief over what value *latent variables*/model parameters might take on. *Likelihoods* are also known beforehand representing how data is assumed to be generated given the latent variable for which the *posterior* is to be inferred. Together, these components form the backbone of a probabilistic model and provide the information required to compute the posterior distribution, which is grounded in **Bayesian theory**.[26]

Bayes' Theorem provides the foundational framework for probabilistic reasoning under uncertainty. It describes how to update prior beliefs about unknown quantities - known as latent variables - based on new evidence, resulting in a posterior distribution. Formally, it states that the posterior is proportional to the product of the prior and the likelihood:

$$P(\theta \mid x) \propto P(x \mid \theta) \cdot P(\theta)$$

In probabilistic programming, this principle is applied through models that define prior distributions over latent variables and likelihood functions for observed data. The probabilistic programming language then applies Bayes' Theorem, typically via automated inference algorithms, to compute or approximate the posterior. This makes it possible to express complex probabilistic models declaratively, while delegating the computation of posterior beliefs to the underlying inference engine.

4.2 Inference and Sampling Methods

Inference is the process by which probabilistic models are "executed". Given a model and observed data, the task is to compute the posterior distributions of latent variables. Because probabilistic models often include uncertainty and hidden variables, inference is the process that turns the model into concrete answers. This process is abstracted and automated, but remains computationally nontrivial. Standard approaches to inference are generally grouped into three main categories- *exact inference*, *sampling-based methods*, and *variational inference*.[6, 7, 18].

1. Exact Inference

Exact inference computes the posterior distribution analytically or through complete enumeration. This is possible only in models with limited complexity where the entire state space can be traversed. Although conceptually straightforward, exact inference quickly becomes infeasible as model size increases, due to the combinatorial explosion of possible states.

2. Sampling-Based Inference

Sampling-based inference approximates the posterior by drawing samples from it, typically using Markov Chain Monte Carlo (MCMC) methods. Continuously, random samples are generated from a given distribution - usually the prior or a conditional- and used to progressively approximate the posterior with the model being executed repeatedly. These approaches are flexible and can handle complex models, but they are often computationally expensive and require many samples to ensure accuracy. The resulting samples can then be used to estimate expectations, marginal distributions, or event probabilities.

3. Variational Inference

Variational inference formulates inference as an optimization problem. Instead of sampling, it produces a family of approximate distributions and seeks the member that is closest to the true posterior. This involves defining an objective function - often the *evidence lower bound (ELBO)* - and optimizing it using gradient-based methods. Variational inference tends to be faster and more scalable than sampling, though it trades off exactness for speed by approximating the posterior rather than sampling from it directly [7].

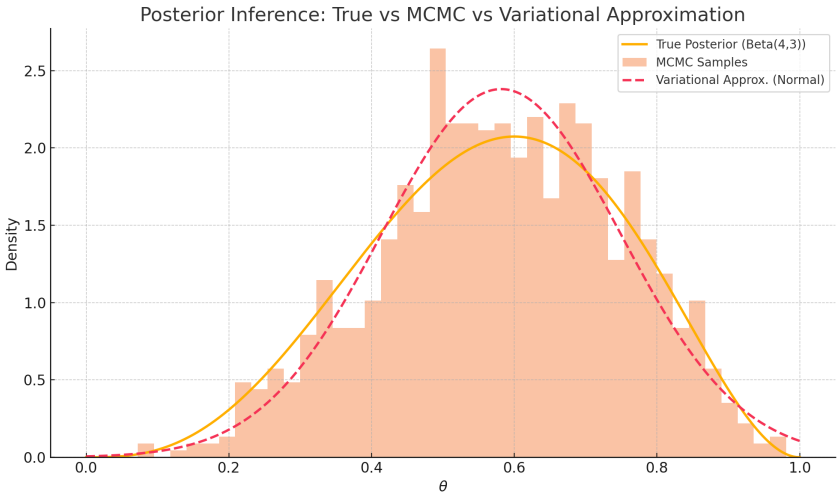


Figure 4.1: Comparison sample of exact posterior to inference methods

4.3 Symbolic Semantics and Model Checking

While probabilistic programming is typically associated with statistical inference and sampling-based evaluation, there exists an alternative perspective that treats probabilistic programs as formal models subject to verification [29, 31] as described in section 3.1 . In contrast to traditional inference, symbolic evaluation and model checking aim to rigorously analyze all possible executions of a program against formal specifications.

Symbolic approaches differ in that they do not enumerate individual executions or samples, but instead reason over entire sets of possible states and paths using abstract representations. This enables analysis of program behavior in a more in-depth and automated way, making it possible to answer questions like: "Does this property always hold, regardless of sampling outcomes?" or "What is the probability that this condition is eventually satisfied?"

Such techniques are particularly relevant for verifying probabilistic systems where correctness, safety, or reachability must be guaranteed rather than estimated. They also enable reasoning about programs that would be challenging to analyze through sampling alone, such as those involving recursion, nondeterminism, or infinite state spaces.

A growing body of foundational work has explored how symbolic and model checking techniques can be adapted to probabilistic settings [5, 15, 16]. This includes approaches based on probabilistic automata, logics for probabilistic reasoning, and symbolic representations of state spaces.

Building upon these foundations, the POPACheck tool [30] represents a pioneering approach to verifying probabilistic programs with recursive structure. By translating programs into probabilistic operator precedence automata - a formal model that can represent infinite state spaces - POPACheck allows for the analysis of properties expressed in temporal logics. This enables verification of systems that lie beyond the scope of most traditional probabilistic programming frameworks.

In this context, *MiniProb* is used not for inference, but as a front-end language for describing probabilistic behavior that is ultimately verified through model checking. The distinction between sampling-based execution and symbolic verification is presented below.

4.4 Methodological Comparison

Table 4.1: Comparison of Probabilistic Programming in Traditional Inference vs. Formal Verification

Aspect	Traditional Inference (e.g., Bayesian Modeling)	Formal Verification (e.g., Model Checking)
Primary Goal	Estimate posterior distributions over variables (e.g., $P(\theta \mid data)$)	Prove correctness properties about probabilistic behavior (e.g., reachability, safety)
Evaluation Strategy	Sampling-based execution (e.g., MCMC, SMC, VI)	Symbolic or exhaustive exploration of probabilistic state space
Execution Semantics	Program runs as a simulator; samples are drawn during execution	Program is interpreted mathematically; all behaviors are analyzed systematically
Handling of Randomness	Uses actual draws from distributions during execution	Treats random variables symbolically or via full enumeration
Output	Posterior samples, estimates, and confidence intervals	Probabilistic bounds, expected values, or formally verified properties
Typical Model Assumptions	Often uses continuous distributions and models real-world noise	Often restricted to discrete, well-defined models for tractability
Result Guarantees	Approximate; dependent on convergence and diagnostics	Sound (and often complete) within model and logic constraints

The *MiniProb* DSL

5.1 Domain and Purpose

MiniProb is the successor of a DSL first introduced as *MiniProc*[11]. Originally lacking the typical probabilistic primitives such as `observe` and `query`, the language was designed for formal verification of recursive programs (see 3.1). As MiniProc focuses on recursive control-flow and call-return semantics - qualities representative of procedural programming - it classifies under the imperative paradigm (see 2.2.2). Furthermore, it is a context-free language, since its structure is tightly coupled with the theory of **Operator Precedence Languages** (**OPLs**), a deterministic subclass of CFLs (see: 2.1). Programs written in MiniProc can be directly compiled into **Operator Precedence Automata** (**OPA**), which leverage the structured nesting of procedure calls and returns, thus enabling automata-based model checking via the *POMC* tool against specifications written in the Precedence-Oriented Temporal Logic (POTL) [11]. This conversions is made possible because MiniProc constraints its syntax to comply with the *precedence relations* defined by a given **Operator Precedence Matrix** (**OPM**), allowing for deterministic and unambiguous parsing.

MiniProb extends MiniProc by incorporating primitives from PPLs [30], which enable sampling, conditioning and nested inference (see 3.3). While remaining recursive-procedural, it now belongs to the family of probabilistic programming languages, still within the context-free class, but no longer parsable to an OPA. However, with the introduction of non-determinism and probabilistic transitions, MiniProb programs are able to be translated into **probabilistic Operator Precedence Automata (pOPA)** [30, 29], that can represent both stack and stochastic behavior. This transition enables formal verification of MiniProb programs via model checking, supporting *qualitative*, *quantitative*, and *approximate* queries, allowing to check whether properties - specified in $POTL f_\chi$ - hold with certainty, compute exact satisfaction probabilities, or obtain estimated bounds when exact computation is infeasible [12].

> POTL is a temporal logic specifically designed to express properties over Operator Precedence Languages (OPLs) [10, 9]. Extending LTL, it includes modalities that refer to the hierarchical structure induced by procedure calls and returns, enabling formal reasoning about context-sensitive behaviors such as matching call-return sequences, stack inspection, and structured exception handling. The fragment POTL_{f_χ} restricts the logic to a form suitable for model checking over nondeterministic and probabilistic systems - particularly when used with probabilistic Operator Precedence Automata (pOPAs) - while retaining the expressive power needed to reason about structured control flows [29].

MiniProb also incorporates expressing higher-order reasoning patterns, such as those involving mutually recursive queries that simulate agents reasoning about each other's beliefs or decisions - a feature common in cognitive modeling and multi-agent systems [36].

As a DSL, MiniProb is not intended to offer the general-purpose capabilities of mainstream languages. In particular, it is not Turing-complete: it lacks features like unbounded memory manipulation, arbitrary recursion without termination guarantees, or dynamic data structures. Its constructs are deliberately limited to ensure compatibility with pOPA translation and to preserve decidability of model checking, which would not be possible in a fully general computational model.

Ultimately, the purpose behind both MiniProc and MiniProb is to offer a high-level, user-friendly interface for programmers wishing to verify properties of recursive (probabilistic) programs without needing to manually translate their source code into OPA or pOPA representations. These DSLs thus serve as a crucial bridge between real-world programming constructs and the formal models required for rigorous verification.

5.2 Syntax

$$
\begin{aligned}
\mathit{prog} &\rightarrow (\mathit{decl}\ \texttt{;})^*\ \mathit{func}^+ \\
\mathit{decl} &\rightarrow \mathit{type}\ \mathit{id}\ (\texttt{,}\ \mathit{id})^* \\
\mathit{func} &\rightarrow \mathit{id}\texttt{(}(\mathit{type}\ \texttt{\&}?\ \mathit{id}(\texttt{,}\ \mathit{type}\ \texttt{\&}?\ \mathit{id})^*\ )?\texttt{)} \\
&\quad\ \texttt{\{}(\mathit{decl}\ \texttt{;})^*\ \mathit{stmt}^+\texttt{\}} \\
\mathit{fcall} &\rightarrow \mathit{id}\ \texttt{(}[\mathit{e}\ (\texttt{,}\ \mathit{e})^*\ ]\texttt{)} \\
\mathit{stmt} &\rightarrow \mathit{lval}\ \texttt{=}\ \mathit{e}\ \texttt{;} \\
&\quad\ |\ \ \mathit{lval}\ \texttt{=}\ \texttt{Distribution(...) ;} \\
&\quad\ |\ \ (\texttt{query})?\ \mathit{fcall}\ \texttt{;} \\
&\quad\ |\ \ \texttt{observe}\ \mathit{e}\ \texttt{;} \\
&\quad\ |\ \ \texttt{throw ;} \\
&\quad\ |\ \ \texttt{if (}\mathit{e}\texttt{) \{}\ \mathit{stmt}^*\ \texttt{\} else \{}\ \mathit{stmt}^*\ \texttt{\}} \\
&\quad\ |\ \ \texttt{while (}\mathit{e}\texttt{) \{}\ \mathit{stmt}^*\ \texttt{\}} \\
&\quad\ |\ \ \texttt{try \{}\ \mathit{stmt}^*\ \texttt{\} catch \{}\ \mathit{stmt}^*\ \texttt{\}} \\
\mathit{lval} &\rightarrow \mathit{id}\ (\texttt{[}\mathit{e}\texttt{]})? \\
\mathit{type} &\rightarrow \texttt{bool}\ \mid\ (\mathit{s}\mid\mathit{u})\ \mathit{int}\ (\texttt{[}\mathit{int}\texttt{]})?
\end{aligned}
$$

Figure 5.1: Compacted grammar of `MiniProb`.

A compacted grammar for MiniProb is shown in 5.1 containing the core elements for intuitive comprehension. A full BNF grammar can be found in the appendix, see A.1. A *prog*ram consists of zero or more top-level declarations followed by one or more function definitions. Each *decl*aration introduces one or more variables of a given *type*, which may be `bool`, signed (`s`) or unsigned (`u`) integers of certain bitwidth (e.g., `s8`, `u16`), and optionally fixed-size arrays (e.g., `u8[10u4]`). Functions (*func*) consist of an identifier, an optional list of typed parameters (passed by value or reference via `&`), and a block containing optional local declarations and at least one statement.

Within function bodies, MiniProb allows for local declarations and familiar C-like statements, including variable assignments (`lval = e;`), function calls, and control flow structures such as `if/else`,

`while` loops, and structured exception handling via `try/catch`, with `throw` used to abort execution. Left-hand values (*lval*) may refer to scalar variables or indexed array elements, while types follow a compact convention, consisting of Booleans and signed or unsigned integers of fixed width.

Probabilistic constructs are central to MiniProb's design. These include sampling from distributions via statements like `lval = Distribution(...);`, currently supporting both `Bernoulli` and `Uniform` distributions. The `observe e;` construct allows conditioning execution on logical expressions, rejecting traces that do not satisfy the observed condition. Inference is triggered through the `query fcall;` statement, which indicates the probability evaluation of a function's outcome given a set of observations.

In addition to structured control, MiniProb supports expressions built from literals, variables, and function calls, composed using arithmetic operations (`+`, `-`, `*`, `/`, `\%`), comparison operators (`==`, `!=`, `<`, `<=`, etc.), and Boolean connectives (`!`, `\&\&`, `||`). These expressions can appear on the right-hand side of assignments or as arguments.

Beyond the standards, MiniProb introduces *probabilistic assignments* to express uncertain choices: $lval = e_1\{e_2 : e_3\}\ e_4$. This construct selects e_1 with probability $p = e_2/e_3$, and e_4 with probability $1 - p$. Multiple such choice blocks can be chained to define multi-way probabilistic decisions, enabling a compact encoding of stochastic control.

Identifiers follow the pattern `[a-zA-Z\_][a-zA-Z0-9.\_\:$\sim $]*`, allowing names with symbols like underscores, colons, dots, and tildes. Integer literals are annotated explicitly with both value and type, e.g., `42s8`, `7u16`, where the suffix denotes signedness and bit width. Boolean literals are written plainly as `true` and `false`.

Lastly, MiniProb also includes a few more constructs which are omitted here as, they are currently not covered by the language extension. These constructs include `probabilistic query: ...` defining the type of query, `formulas = ...` - specified by POTL formalisms - and the definition of *modules* [12].

5.3 Semantics and Example Models

To support reliable analysis and ensure well-formed behavior, MiniProb enforces a few core semantic rules. Each program must include a `main` function with no parameters. When passing arguments by reference (value-result), the actual argument must be a variable identifier rather than a literal or expression. Additionally, the `query` construct can only be used on functions that declare at least one reference parameter.

These rules are exemplified in the following implementation of a simplified *Schelling coordination model*. In this setup, two agents - Alice and Bob - independently choose between two cafés, preferring to coordinate without direct communication. Each agent samples an initial preference, then recursively queries the other's decision to adjust its own. The model uses probabilistic sampling and mutual `query` calls, while `observe` enforces alignment between the agents' choices. The shared variable `p` bounds recursion depth, ensuring termination.

```
program:
u4 p;
main() {
  bool res;
  p = 11u4;
  query alice(res);
  // res is which café they have gone to
}

alice(bool &x) {
    bool prior_alice, bob_choice;
    // sample according to the prior (0.55)
    prior_alice = 1u1 {11u5 : 20u5} 0u1;
    p = p - 1u4;
    query bob(bob_choice);
    observe prior_alice == bob_choice;
    x = prior_alice;
}

bob(bool &y) {
    bool prior_bob, alice_choice;
    // sample according to the prior (0.55)
    prior_bob = 1u1 {11u5 : 20u5} 0u1;
    if (p > 0u4) {
        query alice(alice_choice);
        observe prior_bob == alice_choice;
    } else {}
    y = prior_bob;
}
```

Figure 5.2: MiniProb implementation of the Schelling coordination model.

Part II

Design & Implementation

Requirements and Technological Context

As discussed in in the introduction 1.1, the main aim is the development of a VS Code Language extension that operates on par with well-established tooling for other programming languages. The following requirements define the expected capabilities and characteristics the extension must demonstrate.

Functional Requirements

1. The extension must provide syntax highlighting for the target language, differentiating keywords, operators, literals, and identifiers.
2. The extension must provide basic code completion suggestions based on language keywords and the abstract syntax tree (AST) generated by Langium.
3. The extension must be able to validate data types within the language, providing warnings or errors when type mismatches occur.
4. The extension must perform semantic analyses to detect invalid constructs, undefined variables, and other semantic errors.
5. The extension must display syntax and semantic error messages in the editor in real-time as the user edits the code.
6. The extension must utilize the Langium framework for language specification, parser generation, and language service implementation.

Non-Functional Requirements

1. The extension must respond to user edits with syntax and semantic feedback within 100-250ms for files of average size.
2. The extension must provide a user-friendly experience that is consistent with other popular VS Code extensions.
3. The extension must be structured and commented in a way that allows future developers to understand and extend its functionality easily.
4. The extension must be compatible with the latest version of Visual Studio Code at the time of its release and be developed using Langium, with consideration for future maintenance and version support.
5. The extension must be able to handle files of immense size (around 10000 lines) of code within the bounds of 1 second.
6. The extension must be delivered with clear and comprehensive documentation, including installation, usage, and extension of its features.

6.1 Alternative Technologies

This extension is implemented using **Langium**, which itself builds upon the standardized **Language Server Protocol** (**LSP**) [10], yet there are several other technologies and approaches available for creating Visual Studio Code language extensions. At its core, the LSP allows tooling for programming languages to be developed in a modular, editor-independent manner, making it possible to implement a language server in any language that can communicate via JSON-RPC.

One common approach is to implement a language server using general-purpose programming languages (e.g., TypeScript, Java, or Rust) combined with parser generation libraries. Technologies such as **Jison** [5], **Nearley** [12], and **PEG.js** [13] offer a range of options for specifying a language's grammar and creating a parser. These libraries vary in design and capabilities and may need extra components, such as dedicated lexers and scoping to enable fully-fletched tooling.

In contrast, more comprehensive technologies such as **ANTLR** [1] and **Langium** go beyond basic parsing. ANTLR provides a mature and highly optimized lexer and parser generation framework that is widely used across many programming languages, while Langium offers an end-to-end solution for language engineering -

including built-in support for scoping, indexing, and validation - making it especially suited for creating fully featured language servers.

For basic syntax highlighting and lexical analysis, **TextMate**-style grammars [15] can be used, offering a lightweight approach when deep semantic understanding is not required. Meanwhile, incremental parsing frameworks such as **Tree-sitter** [16] have gained popularity for their ability to provide high-performance syntax parsing and error recovery, making them ideal for tooling that operates on very large files.

Langium

Langium is a young language engineering framework inspired by the accomplishments of `Xtext`, striving to evolve into a similarly comprehensive and widely-used technology stack.[7] Its goal is to lower the barrier to entry for language design and implementation, opening the field to new users and fostering the growth of its user base. To achieve this, Langium embraces the extensibility of the Visual Studio Code ecosystem and its extension pipeline, provides a rich set of built-in capabilities while strongly focusing on quality and maintainability, ensuring that newcomers and experienced developers alike can build robust, sustainable language tooling.

Although Langium is capable of defining general-purpose languages, its design is primarily optimized for domain-specific languages (DSLs), targeting languages with a limited range of abstraction and complexity. Creating a fully featured GPL is a highly exhaustive endeavor that demands extensive semantic modeling, thorough tooling, and a substantial team of developers. In contrast, Langium is tailored to lower the entry barrier for language design, making it an ideal choice for building DSLs that solve specific, well-defined problems.

Technical Details & Features

Langium is an open-source framework for building programming languages and DSL tooling, distributed under the MIT license and backed by the Eclipse Foundation [8]. Its open-source nature encourages collaboration and continuous improvement through active engagement with the community.

- At its core, Langium builds upon `Chevrotain`, a customizable JavaScript parser engine that is highly optimized. Chevrotain constructs an internal prediction DFA [14] from the grammar's production rules to implement its configurable LL(k) lookahead mechanism, allowing the lookahead depth k to be tuned for both performance and expressiveness. Moreover, Langium exposes this configurability directly - authors can adjust the parser's `maxLookahead` setting to manage grammar complexity and parsing speed. After parsing, Chevrotain produces a Concrete Syntax Tree (CST), which Langium then converts into a typed Abstract Syntax Tree (AST) using the visitor pattern: by extending Chevrotain's own Visitor interface, Langium traverses the CST and emits AST nodes, preserving semantic structure while discarding syntactic noise. Langium does not incrementally parse documents (i.e., it does not reuse prior parse results to update only modified regions), yet it still offers significant performance benefits compared to alternative engines [3], complementing its generation of precise ASTs and CSTs and its robust error recovery capabilities. The error recovery is implemented with single-token insertion and deletion strategies to correct unexpected or missing tokens, alongside rule-level resynchronization that discards input until a defined recovery point is reached, to maintain parser continuity in the face of syntax errors.
- Langium provides a custom DSL using the `.langium` extension to define the grammar on which everything else is built upon. This DSL closely resembles an EBNF grammar, augmenting basic BNF with regex-like operators, language constructs for assignments and typing, and built-in cross-referencing. Cross-referencing enables production rules to declare relations to one another, which are resolved during a document's linking phase. Once defined, the core grammar is converted into a Chevrotain compatible JSON view, which when consumed by the parser, produces a fully typed

AST. While linking, the AST is traversed and any cross-references resolved by checking the reference against the computed scope of each possible reference holder.

- In addition to its core language capabilities, Langium offers further extensibility: its generator API traverses ASTs and CSTs to produce arbitrary outputs - whether transpiled or compiled code, structured data formats such as JSON, XML, or Markdown, or custom domain-specific representations - and thus supports downstream processing, tooling, or integration into build and deployment pipelines. Langium projects can also be scaffolded into a command-line interface, registering commands for parsing, validation, generation, and LSP serving; each command invokes the corresponding language services to parse source files, apply validation rules, execute generator or interpreter routines, and emit results, thereby enabling a cohesive suite of tools, that leverage Langium's core capabilities across diverse environments.
- Leveraging established and widely adopted ecosystems is central to the design of Langium, allowing it to benefit from mature tools, protocols, and conventions. As a result, it adopts the LSP - standardizing language tooling across platforms and environments - to enable seamless integration with editors and IDEs such as Visual Studio Code and Eclipse Theia. In the same vein, it is developed in TypeScript and distributed via npm, which allows it to build upon the robust tooling, typing support, and package infrastructure of the JavaScript and TypeScript landscapes.
- To further streamline the DSL development process, Langium provides a dedicated scaffolding tool based on `Yeoman`. This generator automates the creation of a fully configured project, including a grammar template, build pipelines, and example code for parsers, validations, and editors. Removing boilerplate setup provides a standardized structure to build upon and saves developers significant time and effort, making Langium a practical and accessible platform.

7.1 High-Level Architecture Diagram & Module Decomposition

Langium projects can easily be created with the support of the scaffolding tool `Yeoman`, producing a structured mock project including a sample language definition. Most of the definitions are found in the `src` directory 7.1, encompassing functionality for the Language Server, the extension and actual language tooling.

`language/` Assembles the core of any Langium project. It includes the `.langium` file, in which the custom language grammar is defined, as well as the code that starts the Language Server with all necessary services bundled.

`cli/` Provides functionality for CLI interactions, command parsing, and generators. The generators in this folder invoke processes that generate artifacts from the language and input files; they are typically used only offline and are called once rather than continuously via the CLI and support efficient pipelining.

`extension/` Contains the VS Code extension bootstrap code needed to connect to the Language Server and utilize LSP support features. All mentioned folders except the one containing CLI logic are bundled into the extension itself.

`generated/` Contains artifacts produced from the grammar definition in the `.langium` file. This includes TypeScript AST type definitions, which are essential for further analysis and validation, as well as the preliminary parsed grammar rules formatted for the Chevrotain engine to generate parsers.

Services The Services in Langium are a collection of utility classes and helper functions offered to every component of a language project via dependency injection. They cover an extensive range of functionality, from LSP-specific features - such as providers for context-aware code completion, rich hover tooltips, signature help, go-to-definition and find-references - to core language utilities that expose parsed documents as both AST and CST models, manage workspace state and context information (including indexing, scoping, and name resolution), synchronize the in-memory document model and perform validation

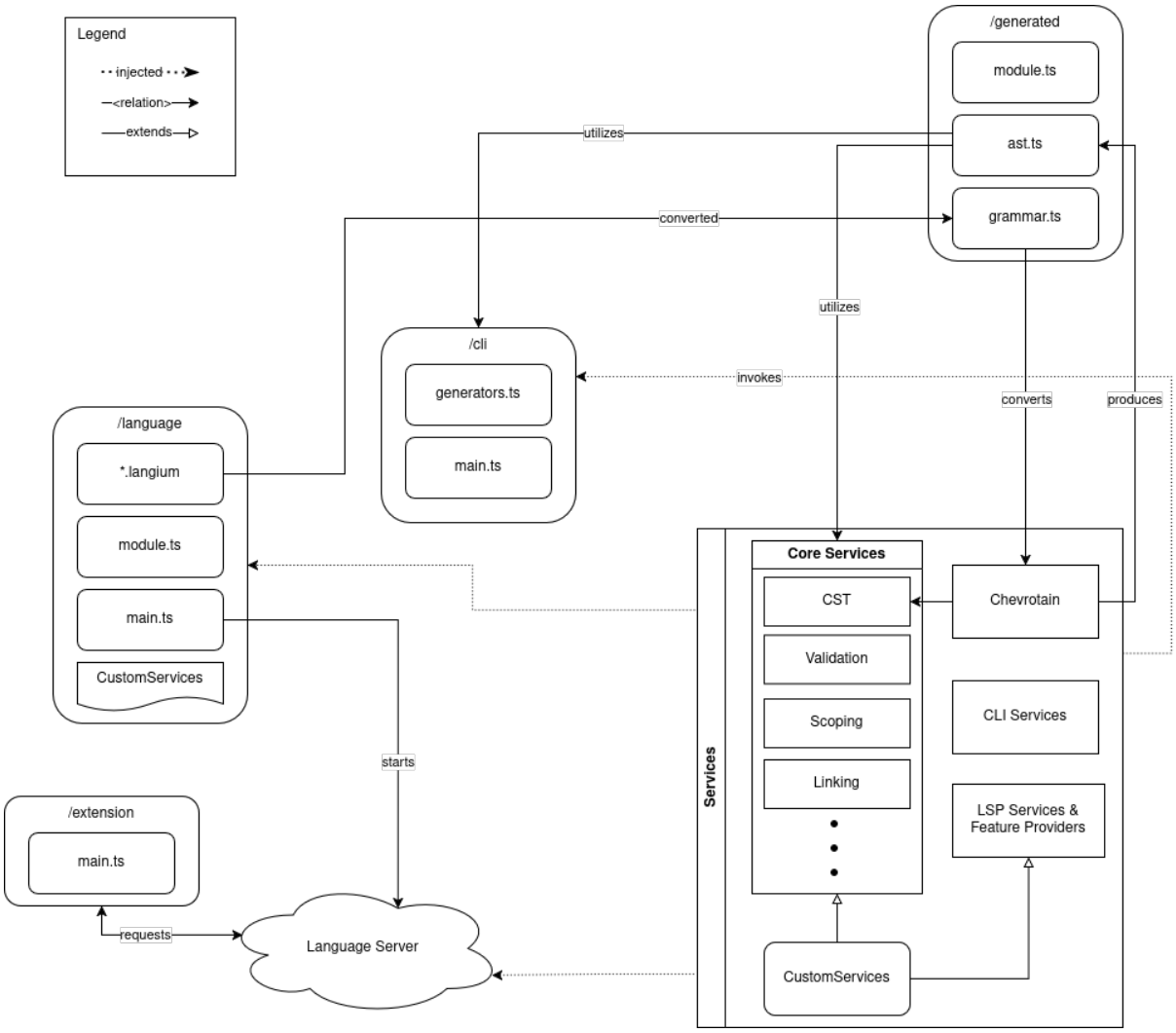


Figure 7.1: Architecture overview of a Langium project

checks. Because the parts are wired up as an injectable service, precise consumptions of the capabilities needed is possible. Moreover, Langium enables developers to swap in custom implementations of services by extending the corresponding default classes, and these services have to be wired into the DI container manually 8.3. By default, the validation logic is located in the `language/` directory; subsequently, I placed additional custom services there. However, as these services grow in both number and complexity, it may be more appropriate to organize them in a dedicated module or folder.

Both `module` files share analogous responsibilities by encapsulating dependency-injection setup and service registration for the language environment. The generated module is produced automatically from the grammar definition and offers two DI fragments, one that registers shared-core services such as AST reflection and another that configures the parser, the Chevrotain grammar factory and language metadata. In contrast the main module is handwritten and imports Langium's default core and shared fragments, registers any custom services such as validators, scope providers and caches, invokes `inject` to assemble the full service container and exposes a `createServices` factory.

Although the core logic resides in `/src`, other files of interest reside at the project root including configurations for npm/node and Langium itself: `language-configuration.json`, which governs editor behaviors such as comment delimiters, bracket pairs and auto-closing rules, and `langium-config.json`, which declares settings like the language ID, grammar entry point, target file extensions and maximum lookahead. In particular, the `syntaxes` config property enables the generation of either a '.monarch.ts' (for Monaco-based editors) and a '.tmLanguage.json' file (for TextMate-based engines). The TextMate grammar maps regex patterns to semantic scopes that underpin VS Code's coloring, bracket matching and code folding, and is the one consumed by the extension. Customizing the syntax files allows developers to implement nuanced highlighting - such as semantic theming and advanced pattern recognition - that cannot be exhaustively defined by grammar rules alone [4].

Notably, projects initialized with the scaffolding tool include extra files not discussed; these include, amongst others, the Monarch grammar files, Vite-related configs, and setup scripts - scaffolded for a web demo but can be removed when solely focusing on developing the language tooling extension.

7.2 Data Flow and Control Flow

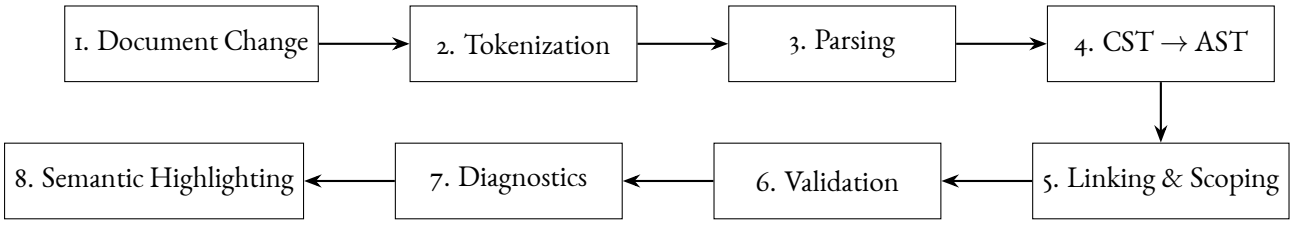


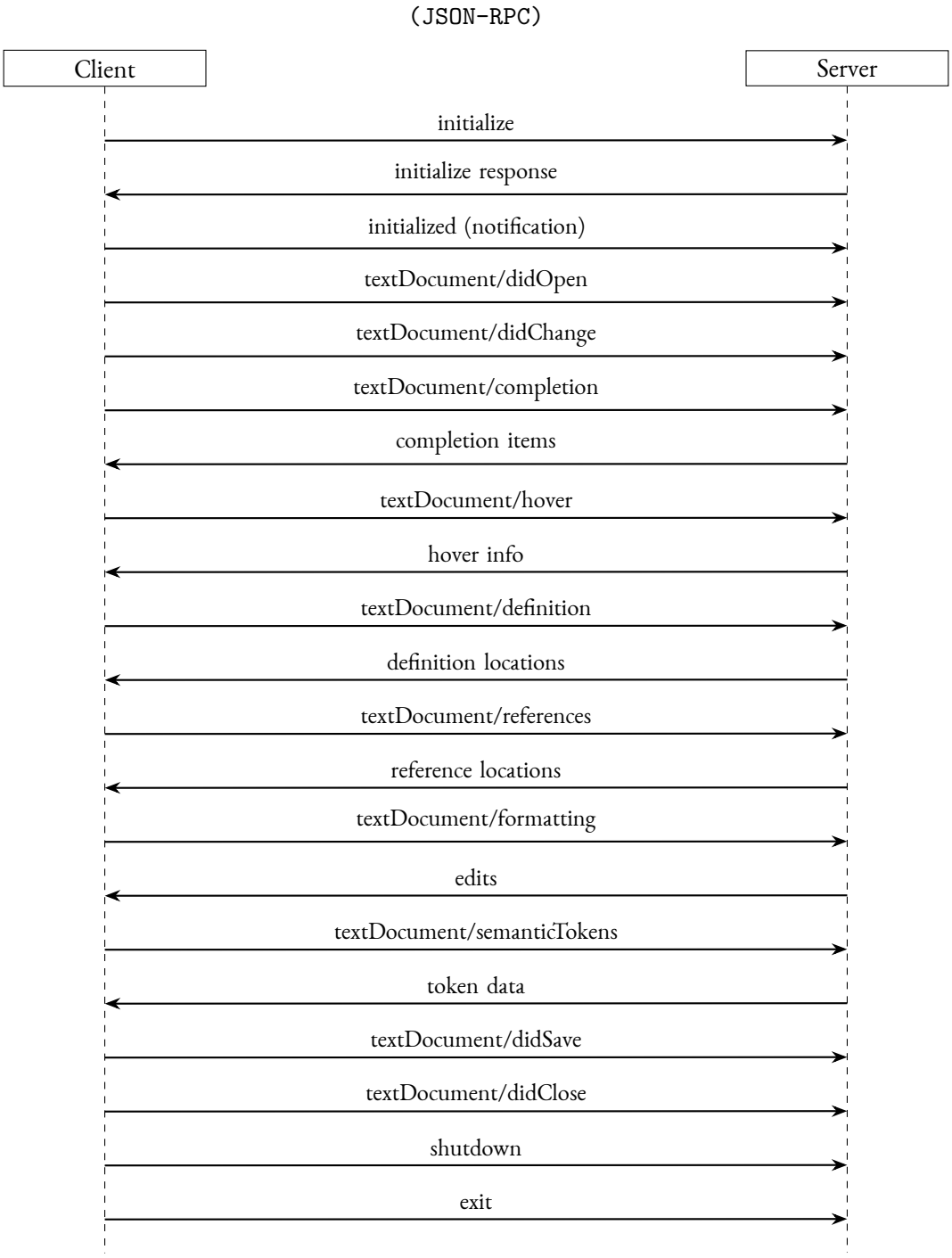
Figure 7.2: Flow chart of Langium's document process pipeline

The editor (via VS Code's LSP client) detects file opens or edits and sends an `onDidChangeTextDocument` notification to the server [9]. Chevrotain's Lexer then performs lexical analysis, breaking the raw character stream into a sequence of tokens [2], which the Chevrotain parser consumes to build a Concrete Syntax Tree (CST). Langium's AST Builder walks the CST to produce a higher-level Abstract Syntax Tree (AST) [6], after which the `ScopeProvider` and `Linker` resolve cross-references, the `ValidationRegistry` applies semantic rules to generate diagnostics, and the server finally publishes diagnostics, semantic tokens, completions, hover information, and other language-service results back to the editor via the Language Server Protocol.

Language Server Protocol: Server client interaction

This illustration is a concise comprehension aid to show how the Language Server Protocol enables standardized communication between an editor (client) and a language server: from the initialization handshake through document events, feature requests (completion, hover, definition, references, formatting, semantic tokens), and shutdown [11]. By defining a common set of well-specified messages and payloads, LSP serves as a generally

applicable protocol that any editor and any language server can adopt to instantly share rich tooling capabilities without per-language or per-IDE custom integrations.



Implementation Details

8.1 Langium Grammar Definition for *MiniProb*

A Langium grammar definition primarily consists of rules: `<Rulename>: <production_result>` and terminals: `terminal <name>: <result>` defining the language in accordance of conventional grammatical specifications. In addition, it must also include keywords specifying the grammar name also used in

the AST: `grammar <name>`, and to determine the start from which a document should be parsed: `entry <RootRulename>: <production_result>`. In this case, the production result describes the whole parsable document and therefore acts as the root node of the AST.
Also mention the LL (2) parsability and which tokens demand a 2 lookahead () The code excerpts in the following sections are all taken from the original MiniProb Langium grammar (B.1) and, for easier digestion, have been truncated; each snippet focuses respectively on the core rules, the expression system, and the terminal and type definitions.

Core rules

The code excerpt shown here defines 'MiniProb' as the grammar name and sets the `Program` rule as the entry point, adhering to nominal abstraction by referring to each MiniProb-parsed document as a program. Lines of interest include line 8 - showing that every MiniProb program consists of some global declarations followed by at least on function - and line 4, which allows for file import of other MiniProb programs at the beginning of the document.
The `FileImport` rule emulates the C preprocessor's "#include" directive by requiring a literal match of the `##include` keyword followed by a file name token (terminal `STRING`). Generally during parsing, each line is evaluated either against these fixed string keywords or against any matching terminals (order of match). As the file names have to be retrievable once it is time to compute the scope 8.3, they are made accessible through the AST , with each FileImport-node containing a `file` property that stores the corresponding value.
This declaring of *assignments* (here also referred to as fields) is a central feature of the Langium DSL, which binds other grammar rules or literal strings for subsequent retrieval and use in AST-based computations. Because all production rules extend upon the `AstNode` base class, the assigned fields share a common origin allowing polymorphic access and seamless traversal and processing of every rule-node; the exception being string literals, which are treated as language specific keywords (eg. "while", "throw"). Fields can either be single valued (line 24) or multi-valued (line 12) in which case an array of the referred rule is created instead of a single instance. Furthermore, fields may be assigned optional rules, which are backed in the AST by nullable values (lines 5, 6, 16), or if multi-valued, by types that default to undefined (lines 4, 8). It is also possible to directly mark assignments as optional (line 20), if so, the AST denotes the property as a boolean flag indicating whether the field was present.
Bracketed annotations (`[<Rule>:<Value>]`, line 16) impose stronger cross-referential constraints by requiring that the target symbols be defined elsewhere in the document/scope and hold a field with the same value; such cross-references are resolved during the scoping phase (see Figure ??). When a field references multiple distinct rules, the `type` keyword has to be used (line 2) unifying the rules into a single symbol, which generates an union type within the AST [17]. In particular, all union members must share the same underlying value type: consider line 16, 12 and 20, Lval refers to either Param or Decl expecting the resolution to occur over an ID.

```
1  grammar MiniProb
2  type DeclOrParam = Decl | Param;
3  entry Program:
4      (fileImports+=FileImport)*
5      probabilisticQuery=PROB_QUERY? // not tested: out of scope
6      formula=FORMULA? // not tested: out of scope
7      'program:'?
8      (declarations+=Decl ';')* functions+=Func+;
9  FileImport:
10     '#include' file=STRING;
11 Decl:
12     type=Type names+=ID (',' names+=ID)*;
13 .
14 .
15 Lval:
16     ref=[DeclOrParam:ID] ('['index=Expression']')?;
17 .
```

```
18  .
19  Param:
20      type=Type byRef?='&'? name=ID;
21  ArgList:
22      arguments+=Arg (',' arguments+=Arg)*;
23  Arg:
24      expression=Expression; //only expression and no by-ref symbol (could be replaced)
25  .
26  .
```

Listing 8.1: Langium grammar core of MiniProb

Expressions

Most of the expression system is omitted here, as it closely follows the *operator-precedence grammar* pattern common to LL parsers [operatorPrecedence]. Typically, a grammar defines a series of nonterminals - each corresponding to a distinct operator-precedence level (from primaries up to full expressions) - and structures its productions so as to eliminate left recursion while preserving left-associative evaluation at every level. This arrangement guarantees that the parser can always choose the correct production with a single lookahead token, ensuring LL(1) compliance.

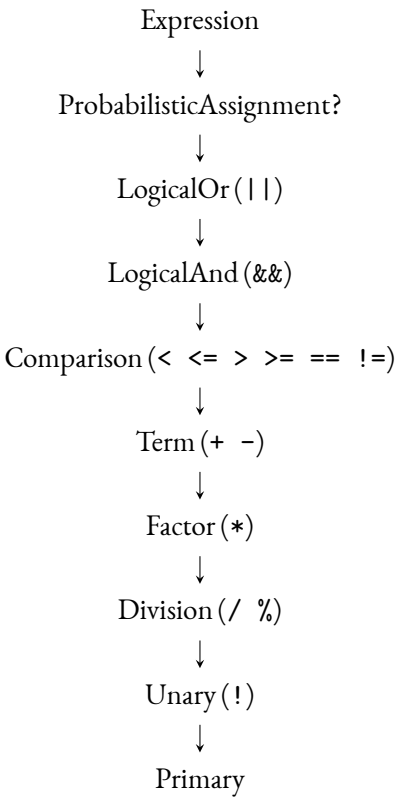


Figure 8.1: Operator-precedence hierarchy

In Langium, rules can be made to share other base classes than `AstNode` by utilizing the `infers` keyword. This is especially useful for creating a layered expression system, as all the levels infer from the top-most expression (line 3), which follows logical abstraction and enables dynamic and easy resolution of parsed expressions. The lines depicted here have been selected because they include the `Probabilistic Assignment`, a construct particular to MiniProb (see 4.4), alongside the `Primary` rule, which defines the core forms to which all expressions ultimately resolve. The topmost level of the precedence hierarchy was originally occupied by the disjunction operator (`||`), but was ultimately edged out by the probabilistic assignment. This behavior is enabled by another Langium specific command `infers`, which instructs a rule to create a new AST node based on the structure of the input. In this case, an `Expression` is by default parsed as a `LogicalOr`, but immediately becomes a `Probabilistic Assignment` node as soon as the first curly brace is recognized (lines 2, 4). The remainder of the operator precedence follows conventional conventions, with the exception that '`*`' binds more loosely than division and modulo (see Figure 8.1).

All expression consist of `Primary`'s on an elementary level and can be either boolean, an integer literal, or a variable reference (`Lval`); treating `Lval` as a `Primary` makes variable accesses first-class expressions, as is conventional in many languages. In addition, the production allows any expression to be wrapped in arbitrarily many parentheses which only delegate the inner expression without altering the meaning.

```
1  ProbChoice:
2      '{'numerator=Expression ':' denominator=Expression '}';
3  Expression:
4      LogicalOr ({infer ProbabilisticAssignment.head=current}
       ↪  (probabilities+=(ProbChoice) fallbacks+=LogicalOr)+)?;
```

```
5  LogicalOr infers Expression:
6  .
7  .
8  .
9  Primary infers Expression:
10     {infer BoolLiteral} literal=BOOL |
11     {infer IntLiteral} literal=IntegerLiteral |
12     Lval |
13     '(' Expression ')';
```

Listing 8.2: Langium grammar expression system of MiniProb

Types and terminals

The excerpt displayed here, defines the types alongside its literal lexical vocabulary. The nonterminal `Type` accepts either the literal keyword `bool` or an `IntType`. An `IntType` specifies if it's signed and of what width and may be converted to an `IntArray` if brackets are present (line 4). The `IntegerLiteral` rule recognizes numeric constants with an optional printable sign. Together, these productions ensure that boolean types, integers, and arrays are adequately represented according to the `MiniProb` definition (4.4).
Lexical terminals and hidden terminals are distinguished to control which tokens contribute to the final model. Terminals defined in lines 8-13 produce concrete tokens that appear in the parse tree as syntactic elements or attribute values; several of these declarations include a `returns` clause (lines 9, 12, 13) to instruct the parser on the semantic type to assign in in the resulting node. Hidden terminals defined in lines 14-16 are recognized by the lexer but discarded before parsing, preventing formatting and commentary from influencing the structural representation.
Moreover, the `ID` terminal employs a *negative lookahead* to exclude specific sequences from being recognized as identifiers (line 11). This mechanism ensures that boolean literals and integer prefixes are not misclassified as identifiers, preserving the distinct classification of reserved words and type prefixes without requiring further enforcements.
Finally, should input match multiple terminal patterns, the lexer resolves ambiguities by first selecting the pattern that consumes the most characters (longest-match rule) and then, in the event of equal length, the rule declared earlier in the grammar.

```
1  Type:
2      'bool' | IntType;
3  IntType:
4      prefix=INT_PREFIX ({infer IntArray} '[' size=INT ']')?;
5  IntegerLiteral:
6      sign=('+' | '-')? value=INT suffix=INT_PREFIX;
7
8  terminal INT_PREFIX: /[su][1-9][0-9]{0,8}/;
9  terminal BOOL returns boolean: /(true|false)/;
10 .
11 terminal ID: /(?!(true|false)|[su][1-9][0-9]*)[a-zA-Z_][a-zA-Z0-9_\.\:\~]*/;
12 terminal STRING returns string: /"(\\.|[^"\\])*"|'(\\.|[^'\\])*'/;
13 terminal INT returns number: /[0-9]+/;
14 hidden terminal WS: /\s+/;
15 hidden terminal ML_COMMENT: /\/\*[\s\S]*?\*\//;
16 hidden terminal SL_COMMENT: /\/\/[^\n\r]*/;
```

Listing 8.3: Terminals and patterns of MiniProb

8.2 Type System and Semantic Checks

Langium offers the option of formalizing semantic checks and type systems through a static validation service which can be registered in its dependency-injection framework. Once registered, this service traverses the abstract syntax tree via a visitor pattern, invoking node-type specific validation routines at each step. The following sections describe the implementation of static semantics and the design of the type system.

Validator

The excerpt below presents the registration function exported from `validator.ts` that consolidates all validation routines for statically verifying the parsed syntax tree. This function integrates those checks into the overall services infrastructure and is subsequently invoked in `module.ts` to initialize the complete set of services. As noted, every AST node is processed during the validation phase. Each rule may declare one or more validation methods (provided as an array). These methods receive two parameters - the node instance itself and a ValidationAcceptor - which is used to produce diagnostics shown in the editor, such as error markers, warnings, and informational messages.

```
  export function registerValidationChecks(services: MiniProbServices) {
  const registry = services.validation.ValidationRegistry;
  const validator = services.validation.MiniProbValidator;
  const checks: ValidationChecks<MiniProbAstType> = {
    Lval: validator.checkArrayAccess,
    FuncCall: validator.checkFunctionCalls,
    Func: validator.checkFunctionDefinitions,
    .
    .
    IntegerLiteral: validator.checkIntegerLiteral,
    Program: validator.checkMainOccurrences
  };
  registry.register(checks, validator);
}
```

Listing 8.4: Validation checks registrations

These validations are AST-driven logical procedures that verify specified conditions on the syntax tree. This means that validation is performed by statically inspecting each node's structure, attributes, and inter-node relationships, without relying on runtime information or any other states. Each procedure either invokes the `ValidationAcceptor` to report deviations when constraints are violated or lets the node pass unremarked when it satisfies the criteria. As an example, consider the following `checkMainOccurrences` method, which enforces the invariant that a `main()` function must be defined exactly once in each program.

```
  checkMainOccurrences(node: Program, accept: ValidationAcceptor) {
     const functionNames = node.functions.filter(f => f.name === "main")
            .map((f) => f.name);
     if (functionNames.length <= 0) {
      accept("error", 'Each program needs a \'main()\' function.', {
        node,
        property: "functions"
      });
     }
     .
     .
}
```

Listing 8.5: Sample of validation process

Beyond basic AST-based checks, the analysis can be extended by incorporating models that augment node-level information with additional contextual or semantic data. In such addons, the AST serves as the seed from which richer representations and semantics are derived and maintained in dedicated structures. This also enables the implementation of a custom type system whose validation routines are registered alongside other checks, allowing the validator to invoke type-checking services during the validation phase.
The type system used here is much simpler than those found in mainstream GPLs and comprises just four files. The `operation.ts` and `compatible.ts` modules each contain targeted logic: for assignments and function calls, `compatible.ts` verifies that the involved types are compatible (the integer corner cases are handled here as well), while for all expressions and operators, `operation.ts` checks that the operand types are valid for the given operation. Both return their judgement as a simple boolean to the validator, which can react accordingly and - already aware of the involved types - produce detailed error messages.
The `description.ts` module defines the core vocabulary of the type system, including a TypeDescription union (e.g. `BooleanTypeDescription`, `IntegerTypeDescription`, etc.), factory functions (`createIntegerType`, `createErrorType`, ...) and predicates (`isIntegerType`, `typeToString`, ...). It also introduces an explicit `ErrorTypeDescription` to represent and propagate type-checking failures without interrupting the overall inference process. The `infer.ts` file leverages these descriptions by traversing the AST: for each node it recursively infers subexpression types - adhering to the system's inference and compatibility rules (see axiomatic rules below)- and returns either a concrete `TypeDescription` or the `ErrorTypeDescription` when conflicts occur, with a simple cache mechanism preventing infinite recursion and boosting performance.
The cache is instantiated anew for each inference run on a node, eliminating any risk of residual data or cross-contamination and ensuring fully deterministic results. However, because it isn't persisted between runs, identical subtrees get re-evaluated for every node, which may introduce redundant work. Unlike incremental parsing systems that retain state to avoid reprocessing unchanged regions, this approach performs a complete, conventional traversal each time. ??try the cache service and see if better???

As an example of how the type system is employed in the validator, this code snippet creates a new cache (a map of `ASTNode -> TypeDescription`) and calls the `inferType` method, which returns the concrete type of the operand. If any inference steps conflict during this process, an `ErrorType` is returned and the propagated message is displayed. Otherwise, the validator verifies that the resolved type is allowed for this operand and handles the error accordingly.

```
checkUnaryExpressions(node: LogicalNegation, accept: ValidationAcceptor) {
  const map = this.getTypeCache();
  const operandType = inferType(node.operand, map);

  if (isErrorType(operandType)) {
    accept('error', operandType.message, { node: operandType.source ?? node });
    return;
  }
  if (!isLegalOperation(node.operator, operandType)) {
    accept(
      'error',
      `The operation '${node.operator}' is not possible on type
      ↪  '${typeToString(operandType)}'`,
      { node, property: 'operand' }
    );
  }
}
```

Listing 8.6: Unary expression validation checks

8.2.1 Semantic and type checks

`checkArrayAccess (Lval)` Infers the index expression's type; reports on conflicting inconsistencies* detected during type inference, or if it is not an integer.

`checkAssignments (Assignment)` Infers both left- and right-hand types (expression or distribution), reports on conflicting inconsistencies* detected during type inference, and finally verifies that the right-hand type is compatible with the left.

`checkFunctionCalls (FuncCall)` Ensures the argument count matches the function's parameter list; for each argument, infers its type and the corresponding parameter's type, reports on conflicting inconsistencies* detected during type inference, checks compatibility, and enforces "value-result" parameters to be of equivalent types.

`checkFunctionDefinitions (Func)` Enforces that a function named `main` has no parameters, and that no two functions in the same `Program` share the same name.

`checkMainOccurrences (Program)` Counts occurrences of functions named `main` and emits an error if there are zero or more than one, ensuring exactly one `main()` per program.

`checkQueryFunctionCall (Query)` Verifies that the referenced function has at least one parameter and that at least one of them is declared `byRef` (value-result).

`checkProbabilisticChoices (ProbChoice)` Infers the types of numerator and denominator, reports on conflicting inconsistencies* detected during type inference, checks that the ':' operation is legal for those types, prevents division by zero, and enforces that the resulting probability lies in $[0, 1]$.

`checkDistributions (Distribution)` Confirms exactly two arguments for each distribution, reports on conflicting inconsistencies* detected during type inference, then ensures both parameters are integers.

`checkBinaryExpressions (BinaryExpression)` Infers left and right operand types, reports on conflicting inconsistencies* detected during type inference, and verifies that the binary operator is legal for those types.

`checkUnaryExpressions (LogicalNegation)` Infers the operand's type, reports on conflicting inconsistencies* detected during type inference, and ensures the logical negation operator is legal for that type.

`checkIntegerLiteral (IntegerLiteral)` Validates that the literal's token text contains no internal spaces, reporting an error otherwise.

`checkDeclarationIds (Decl)` Gathers all declarations in the same scope(see ScopeProvider below), detects duplicate names within the node's own list and against existing declarations, and reports any repeats.

`checkObservationCondition (Observation)` Infers the condition's type and reports an error unless it is boolean.

* Occurring mostly due to mismatching types of sub-expressions.

8.2.2 Axiomatic representation

Base axioms

$$\begin{aligned} \tau &::= \mathsf{Bool} \mid \mathsf{Int}_{s,w} \mid \mathsf{Array}\langle \mathsf{Int}_{s,w} \rangle \quad (s \in \{\top, \bot\},\ 0 < w < 2^{29} - 1) \\ \Gamma &::= \emptyset \mid \Gamma,\, x : \tau \end{aligned}$$

$$\frac{}{\Gamma \vdash \mathit{true} : \mathsf{Bool}}\ (\text{T-True}), \qquad \frac{}{\Gamma \vdash \mathit{false} : \mathsf{Bool}}\ (\text{T-False})$$

$$\frac{}{\Gamma \vdash n^{(s,w)} : \mathsf{Int}_{s,w}}\,(\text{T-IntLit}), \quad \textit{where } s \in \{\top, \bot\}, \quad w \in \mathbb{N}, \quad 1 \le w \le 2^{29} - 1$$

$$\frac{\vdash \mathsf{Int}_{s,w}}{\vdash \mathsf{Array}\langle \mathsf{Int}_{s,w} \rangle}\quad (\text{Ty-Array})$$

Inference rules

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}(\text{T-Var})$$

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash (\, e \,) : \tau}(\text{T-Paren})$$

$$\frac{\Gamma \vdash e : \mathsf{Bool}}{\Gamma \vdash \neg e : \mathsf{Bool}}(\text{T-Not})$$

$$\frac{\Gamma \vdash e_1 : \mathsf{Int}_{s_1,w_1} \quad \Gamma \vdash e_2 : \mathsf{Int}_{s_2,w_2}}{\Gamma \vdash e_1 * e_2 : \mathsf{Int}_{s_1 \vee s_2,\, \max(w_1,w_2)}}(\text{T-Mult})$$

$$\frac{\Gamma \vdash e_1 : \mathsf{Int}_{s_1,w_1} \quad \Gamma \vdash e_2 : \mathsf{Int}_{s_2,w_2}}{\Gamma \vdash e_1 / e_2 : \mathsf{Int}_{s_1 \vee s_2,\, \max(w_1,w_2)}}(\text{T-Div})$$

$$\frac{\Gamma \vdash e_1 : \mathsf{Int}_{s_1,w_1} \quad \Gamma \vdash e_2 : \mathsf{Int}_{s_2,w_2}}{\Gamma \vdash e_1 \bmod e_2 : \mathsf{Int}_{s_1 \vee s_2,\, \min(w_1,w_2)}}(\text{T-Mod})$$

$$\frac{\Gamma \vdash e_1 : \mathsf{Int}_{s_1,w_1} \quad \Gamma \vdash e_2 : \mathsf{Int}_{s_2,w_2}}{\Gamma \vdash e_1 + e_2 : \mathsf{Int}_{s_1 \vee s_2,\, \max(w_1,w_2)}}(\text{T-Add})$$

$$\frac{\Gamma \vdash e_1 : \mathsf{Int}_{s_1,w_1} \quad \Gamma \vdash e_2 : \mathsf{Int}_{s_2,w_2}}{\Gamma \vdash e_1 - e_2 : \mathsf{Int}_{s_1 \vee s_2,\, \max(w_1,w_2)}}(\text{T-Sub})$$

$$\frac{\Gamma \vdash e_1 : \mathsf{Int}_{s,w} \quad \Gamma \vdash e_2 : \mathsf{Int}_{s,w}}{\Gamma \vdash e_1 < e_2 : \mathsf{Bool}}(\text{T-Lt})$$

$$\frac{\Gamma \vdash e_1 : \mathsf{Int}_{s,w} \quad \Gamma \vdash e_2 : \mathsf{Int}_{s,w}}{\Gamma \vdash e_1 <= e_2 : \mathsf{Bool}}(\text{T-Le})$$

$$\frac{\Gamma \vdash e_1 : \mathsf{Int}_{s,w} \quad \Gamma \vdash e_2 : \mathsf{Int}_{s,w}}{\Gamma \vdash e_1 > e_2 : \mathsf{Bool}}(\text{T-Gt})$$

$$\frac{\Gamma \vdash e_1 : \mathsf{Int}_{s,w} \quad \Gamma \vdash e_2 : \mathsf{Int}_{s,w}}{\Gamma \vdash e_1 >= e_2 : \mathsf{Bool}}(\text{T-Ge})$$

$$\frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 == e_2 : \mathsf{Bool}}(\text{T-Eq})$$

$$\frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 \,!= e_2 : \mathsf{Bool}}(\text{T-Neq})$$

$$\frac{\Gamma \vdash e_1 : \mathsf{Bool} \quad \Gamma \vdash e_2 : \mathsf{Bool}}{\Gamma \vdash e_1 \,\&\&\, e_2 : \mathsf{Bool}}(\text{T-And})$$

$$\frac{\Gamma \vdash e_1 : \mathsf{Bool} \quad \Gamma \vdash e_2 : \mathsf{Bool}}{\Gamma \vdash e_1 \,||\, e_2 : \mathsf{Bool}}(\text{T-Or})$$

$$\frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \mathsf{Int}_{s,w} \quad \Gamma \vdash e_3 : \mathsf{Int}_{s,w} \quad \Gamma \vdash e_4 : \tau}{\Gamma \vdash e_1 \{\, e_2 : e_3 \,\}\, e_4 : \tau}\quad (\text{T-ProbAssign})$$

$$\frac{\Gamma \vdash e : \mathsf{Array}\langle \mathsf{Int}_{s,w} \rangle \quad \Gamma \vdash i : \mathsf{Int}_{s,w}}{\Gamma \vdash e[i] : \mathsf{Int}_{s,w}}(\text{T-Index})$$

$$\frac{\Gamma \vdash e_1 : \mathsf{Int}_{s_1,w_1} \quad \Gamma \vdash e_2 : \mathsf{Int}_{s_2,w_2}}{\Gamma \vdash \text{Bernoulli}(e_1, e_2) : \mathsf{Int}_{\bot,1}}(\text{T-Bernoulli})$$

$$\frac{\Gamma \vdash e_1 : \mathsf{Int}_{s_1,w_1} \quad \Gamma \vdash e_2 : \mathsf{Int}_{s_2,w_2}}{\Gamma \vdash \text{Uniform}(e_1, e_2) : \mathsf{Int}_{s_2,w_2}}(\text{T-Uniform})$$

Statement Typing Rules

$$\frac{\Gamma \vdash e_1 : \mathsf{Array}\langle \mathsf{Int}_{s,w} \rangle \quad \Gamma \vdash i : \mathsf{Int}_{s,w} \quad \Gamma \vdash e_2 : \mathsf{Int}_{s,w}}{\Gamma \vdash e_1[i] = e_2 \ \mathsf{ok}}\quad (\text{T-Update})$$

$$\frac{\Gamma \vdash e : \mathsf{Bool} \quad \Gamma \vdash S_1 \ \mathsf{ok} \quad \Gamma \vdash S_2 \ \mathsf{ok}}{\Gamma \vdash \texttt{if}(e)\, S_1 \,\texttt{else}\, S_2 \ \mathsf{ok}}(\text{T-If})$$

$$\frac{\Gamma \vdash e : \mathsf{Bool}}{\Gamma \vdash \texttt{observe}\,(e) \ \mathsf{ok}}\quad (\text{T-Observe})$$

$$\frac{\Gamma \vdash e : \mathsf{Bool} \quad \Gamma \vdash S \ \mathsf{ok}}{\Gamma \vdash \texttt{while}(e)\, S \ \mathsf{ok}}(\text{T-While})$$

$$\frac{\Gamma \vdash e : \tau_e \quad x : \tau \in \Gamma}{\Gamma \vdash x \ \equiv_{comp} \ e \ \mathsf{ok}}\quad (\text{T-Assign})$$

$$\frac{\Gamma \vdash S_1 \ \mathsf{ok} \quad \Gamma \vdash S_2 \ \mathsf{ok}}{\Gamma \vdash \texttt{try}\, S_1 \,\texttt{catch}\, S_2 \ \mathsf{ok}}(\text{T-TryCatch})$$

$$\tau \equiv_{comp} e \quad \text{iff} \quad \begin{cases} \tau_e = \mathsf{Bool},\ \tau = \mathsf{Bool}; \\ \tau_e = \mathsf{Int}_{\top,w_1},\ \tau = \mathsf{Int}_{\top,w_2},\ w_1 \leq w_2; \\ \tau_e = \mathsf{Int}_{\top,w_1},\ \tau = \mathsf{Int}_{\bot,w_2},\ w_1 - 1 \leq w_2; \\ \tau_e = \mathsf{Int}_{\bot,w_1},\ \tau = \mathsf{Int}_{\top,w_2},\ w_1 + 1 \leq w_2; \\ \tau_e = \mathsf{Int}_{\bot,w_1},\ \tau = \mathsf{Int}_{\bot,w_2},\ w_1 \leq w_2; \\ \text{arrays analogous to int} \end{cases}$$

8.3 Custom services

Langium's customization model relies on a straightforward override and extend approach. For each core capability - parsing, validation, indexing, scoping or completion - there is a base service available for replacement or augmentation. Once custom services are registered they integrate seamlessly with all LSP processes such as name resolution, reference linking, code completion and hover, functioning as native parts of the language server.

In this snippet, the project's main `module.ts` file defines `MiniProbModule`, binding `MiniProbValidator` in the `validation` object and registering `ScopeProvider` and `ScopeComputation` factories in the `references` object. The exported `createMiniProbServices` function then calls `inject`, passing the shared context, the generated module, and `MiniProbModule` to populate the `DI` container. Once the `MiniProb` services instance is created, it is registered with `shared.ServiceRegistry.register` (language-agnostic services and infrastructure), and `registerValidationChecks` is invoked to run the validator's own setup routine, completing the integration of the custom services, causing the resolution of validation and reference services to use the custom implementations in place of the defaults.

```
  export const MiniProbModule: Module<
  MiniProbServices,
  PartialLangiumServices & MiniProbAddedServices
> = {
  validation: {
    MiniProbValidator: (services) => new MiniProbValidator(services),
  },
  references: {
    ScopeProvider: (services) => new MiniProbScopeProvider(services),
    ScopeComputation: (services) => new MiniProbScopeComputation(services),
  }
};

export function createMiniProbServices(context: DefaultSharedModuleContext): {
  shared: LangiumSharedServices;
  MiniProb: MiniProbServices;
} {
  .
  const MiniProb = inject(_, MiniProbGeneratedModule, MiniProbModule);
  shared.ServiceRegistry.register(MiniProb);
  registerValidationChecks(MiniProb);
  .
}
```

Listing 8.7: Custom service registration

Scope computation

The `MiniProbScopeComputation` class extends Langium's `DefaultScopeComputation` to enrich the set of exported symbols with user-defined top-level declarations. By overriding the asynchronous `computeExports` method, it first invokes `super.computeExports(document)` to collect the standard exports (primarily global functions). It then retrieves the parsed `Program` AST via `document.parseResult.value` and iterates over each `decl` in `program.declarations` - these `decl`s represent top-level declarations. For every name in `decl.names`, a new `AstNodeDescription` is created and appended to the export list using `this.descriptions.createDescription(decl, name, document)`. This ensures that global declarations - beyond the built-in function exports - are available to the import mechanism and subsequently indexed by the `IndexManager` for cross-file reference resolution.

```
export class MiniProbScopeComputation extends DefaultScopeComputation {
override async computeExports(document: LangiumDocument<AstNode>):
↪  Promise<AstNodeDescription[]> {
  //compute default exports (functions)
  const exports = await super.computeExports(document);

  const program = document.parseResult.value as Program;

  //export top level declarations
  for (const decl of program.declarations ?? []) {
    for (const name of decl.names) {
      exports.push(this.descriptions.createDescription(decl, name, document));
    }
  }

  return exports;
}
}
```

Listing 8.8: Custom ScopeComputation service

Scope provider

The custom scope provider focuses specifically on two kinds of nodes - function calls (`FuncCall`) and variable references (`LVal`) - since these are the only AST nodes in the MiniProb grammar that establish referential links to other symbols. By subclassing the base `ScopeProvider` and overriding its `getScope` method, the provider can dispatch to specialized logic for these two cases while delegating all other node types back to the default implementation. This fallback invocation of `super.getScope()` outside of the `FuncCall` and `LVal` branches ensures that any other or future nodes will receive a fully defined and well-formed scope, preserving completeness of the overall resolution process.

Specialized handling is required because the language supports an import mechanism that selectively brings in only top-level definitions from other files, while excluding all other symbols, which are exported by default into a workspace-wide scope governed by the `IndexManager`. In the `LVal` branch, the provider correctly resolves multiple identifiers declared in a single declarations statement (for example, `u8 i1, i2, i3`) distinguishing between block-scope and global variables, as well as including function parameters as valid resolutions. In the `FuncCall` branch, default behavior is overridden to prevent 'ghost' functions - placeholder symbols automatically invented for unresolved identifiers - and to include in-flight local declarations that appear before their closing delimiters in live editing scenarios (a solution which inspects raw `CST` text and may require future revision).

Because both `FuncCall` and `LVal` reference the same identifier token pattern, `getScope` is invoked for both branches on the same node. If distinguishing syntactic context - such as surrounding parentheses for calls or the presence of an equals sign for assignments - is present, the provider resolves only once aware of what path

to choose.

The helper method below builds a resolution scope for symbols imported from other files. It first transforms each import entry into an absolute URI by joining the current document's directory path with the relative import path. Next, it invokes the `IndexManager` to collect all symbol descriptions of the requested type across the entire workspace, filtering to only those defined in the imported files. Notably, only symbols of already parsed document are managed by the `IndexManager`. To improve performance, the retrieved descriptions are stored in a cache keyed by the document URI and symbol kind, and wrapped in a MapScope for efficient lookup.

```
1  private getImportedScope(
2    fileImports: FileImport[],
3    currentUri: URI,
4    targetNodeType: string
5  ): Stream<AstNodeDescription> {
6    const importUris = fileImports.map((f) => {
7      const filePath = posix.join(dirname(currentUri.path), f.file);
8      return currentUri.with({ path: filePath }).toString();
9    });
10
11   const importKey = 'imported-';
12   if (targetNodeType === Func) {
13     const importedFuncDescriptions = this.descriptionCache.get(
14       currentUri,
15       importKey + targetNodeType,
16       () => new MapScope(this.indexManager.allElements(Func, new
       ↪  Set<string>(importUris)))
17     );
18     return importedFuncDescriptions.getAllElements();
19   }
20   .
21   .
22 }
```

Listing 8.9: Inclusion of imported file's top-level scope

In this snippet, the overridden `getScope` method first confirms that the current resolution request pertains to any `reference` context by checking `context.property` and container presence (lines 1-3), then locates the enclosing program node. Upon detecting a function-call node (line 11), it filters the program's function list to remove placeholder or 'ghost' entries (line 7). A local cache - persistent throughout the whole linking phase - is then used to retrieve or populate descriptors for these real functions (lines 11-22), encapsulating them within Langium's offered scope structure. If file-level imports are present (lines 8, 26-30), a helper routine leverages the global `IndexManager` to fetch additional symbols from imported modules, yielding a secondary stream of descriptors. The in-file and imported descriptors, together with any local declarations, are merged into a composite scope (lines 32-34) that is returned for code completion and reference linking; all other situations fall back to `super.getScope()` (lines 38), preserving the default resolution behavior.

```
1  override getScope(context: ReferenceInfo): Scope {
2    const container = context.container;
3    if (context.property === 'ref' && container) {
4      const program = AstUtils.getContainerOfType(container, isProgram)!;
5
6      // filter Func for body -> only real Func and not ghost Reference(=current
       ↪  input)
7      const programFunctions = program.functions.filter(this.isRealFunc);
```

```
      const includeFileImports = program.fileImports && program.fileImports.length >
      ↪  0;

      var importedDescriptions: Stream<AstNodeDescription> = stream();
      if (isFuncCall(container)) {
        const descriptions = this.descriptionCache
          .get(
            AstUtils.getDocument(container).uri,
            Func,
            () =>
              new MapScope(
                programFunctions.map((func) =>
                  this.astNodeDescriptionProvider.createDescription(func, func.name)
                )
              )
          ).getAllElements();
        .
        .
        //check for imported functions
        if (includeFileImports) {
          const document = AstUtils.getDocument(container);
          const uri = document.uri;
          importedDescriptions = this.getImportedScope(program.fileImports, uri,
          ↪  Func);
        }

        return new MapScope(
          stream(descriptions, importedDescriptions, localDeclarationsDescriptions)
        );
      }
      .
      .
    return super.getScope(context);
}}
```

8.4 VS Code Extension Points

textmate highlighting lang-config.json
??maybe ref from the firt mention of the extension (structure)
The changes made to the files associated with the extension's build are minimal yet essential. The extension's entry point remains `main.ts` in the `extension/` directory, while `tmLanguage.json` serves as the primary configuration for syntax highlighting, specifying the label and groups used to color tokens. During the build pipeline (invoked by `npm run ...`), a small custom script reads `tmLanguage.json`, injects the necessary snippets (8.10) that aren't included by default, and writes the updated file back so that our custom tokens are recognized and styled correctly.

```
  {
    "name": "storage.type.int-prefix.mini-prob",
    "match": "\\b[su][1-9][0-9]*\\b"
  },
  {
    "name": "keyword.control.boolValue.mini-prob",
    "match": "\\b(true|false)\\b"
  }
```

Listing 8.10: Patched syntax configurations

Another critical component is `esbuild.js`, which handles transpiling TypeScript into JavaScript, bundling and minifying dependencies, and resolving imports so that both the "extension main" and "language main" entry points become single `.js` modules. VSCE, the CLI used to package `.vsix` files, requires CommonJS modules, but the repository is set to ESM (`"type": "module"`), so we updated the `package.json` entries in `out/extension/main.js` and `out/language/main.js` to force a CommonJS output. This adjustment not only silenced VSCE's bundling warnings but - by bundling the files through `esbuild.js`-reduced the package footprint from about 27MB of loose output down to roughly 1MB in the final `.vsix`.

8.5 Testing

The Langium project's test suite is initialized via Vitest in the scaffolding-generated environment and is organized into three distinct categories: parsing, linking, and validation. Parsing tests verify that all grammar rules and their associated assignments are correctly recognized and transformed into the appropriate CST and AST node types. Linking tests focus on scope management and reference resolution, ensuring that identifiers and imports are resolved accurately; notably, import resolution tests employ a virtual file system to simulate file paths and confirm that cross-file references function as intended. Validation tests then examine semantic constraints and type correctness, asserting that the constructed AST adheres to the language's typing and business-logic rules. Across all categories, the tests interact directly with CST and AST nodes, making precise assertions that guarantee the language implementation remains consistent and reliable.

8.6 Parsing Speed

conlsuio: mention no dode comülpeteion for structures only refs, not optimal and how it would be implemetned

Foundations

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

[2] Mehrnoosh Askarpour et al. "Formal Methods in Designing Critical Cyber-Physical Systems". In: Oct. 2019, pp. 110–130. ISBN: 978-3-030-30984-8. DOI: 10.1007/978-3-030-30985-5_8.

[3] 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.

[4] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. Vol. 26202649. Jan. 2008. ISBN: 978-0-262-02649-9.

[5] Christel Baier et al. "Symbolic Model Checking for Probabilistic Processes". In: *Proceedings of the 24th International Colloquium on Automata, Languages and Programming (ICALP)*. Vol. 1234. LNCS. Springer, 1997, pp. 61–73. DOI: 10.1007/BFb0035319.

[6] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.

[7] David M Blei, Alp Kucukelbir, and Jon D McAuliffe. "Variational Inference: A Review for Statisticians". In: *Journal of the American Statistical Association* 112.518 (2017), pp. 859–877.

[8] Tevfik Bultan, Richard Gerber, and William Pugh. "Symbolic model checking of infinite state systems using presburger arithmetic". In: *Computer Aided Verification*. Ed. by Orna Grumberg. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, pp. 400–411. ISBN: 978-3-540-69195-2.

[9] Michele Chiari, Dino Mandrioli, and Matteo Pradella. "A First-Order Complete Temporal Logic for Structured Context-Free Languages". In: *Logical Methods in Computer Science* Volume 18, Issue 3 (July 2022). ISSN: 1860-5974. DOI: 10.46298/lmcs-18(3:11)2022. URL: http://dx.doi.org/10.46298/lmcs-18(3:11)2022.

[10] Michele Chiari, Dino Mandrioli, and Matteo Pradella. *POTL: A First-Order Complete Temporal Logic for Operator Precedence Languages*. 2020. arXiv: 1910.09327 [cs.LO]. URL: https://arxiv.org/abs/1910.09327.

[11] Michele Chiari et al. "A Model Checker for Operator Precedence Languages". In: *ACM Trans. Program. Lang. Syst.* 45.3 (Sept. 2023). ISSN: 0164-0925. DOI: 10.1145/3608443. URL: https://doi.org/10.1145/3608443.

[12] Michele Chiari et al. *POMC - A Model Checker for Operator Precedence Languages: User Guide*. https://github.com/michiari/POMC/blob/master/docs/guide.pdf. Accessed: 2025-06-16. Aug. 2021.

[13] Edmund Clarke et al. "Counterexample-Guided Abstraction Refinement". In: *Computer Aided Verification*. Ed. by E. Allen Emerson and Aravinda Prasad Sistla. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 154–169. ISBN: 978-3-540-45047-4.

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

[15] Javier Esparza, Antonín Kučera, and Richard Mayr. "Model Checking Probabilistic Pushdown Automata". In: *Logical Methods in Computer Science* 2.1 (2006), 2:1–2:31. DOI: 10.2168/LMCS-2(1:2)2006.

[16] Kousha Etessami and Mihalis Yannakakis. "Model Checking of Recursive Probabilistic Systems". In: *ACM Transactions on Computational Logic (TOCL)* 13.2 (2012), 12:1–12:40. DOI: 10.1145/2159531.2159534.

[17] Noah D. Goodman. "The principles and practice of probabilistic programming". In: *SIGPLAN Not.* 48.1 (Jan. 2013), pp. 399–402. DOI: 10.1145/2480359.2429117. URL: https://doi.org/10.1145/2480359.2429117.

[18] Noah D. Goodman and Andreas Stuhlmüller. *The Design and Implementation of Probabilistic Programming Languages*. http://dippl.org. Accessed: 2024-01-01. 2014.

[19] Tor Aksel N. Heirung et al. "Stochastic model predictive control — how does it work?" In: *Computers and Chemical Engineering* 114 (2018). FOCAPO/CPC 2017, pp. 158–170. DOI: 10.1016/j.compchemeng.2017.10.026. URL: https://doi.org/10.1016/j.compchemeng.2017.10.026.

[20] Tao Jiang et al. "Formal Grammars and Languages". In: (Nov. 2009). DOI: 10.1201/9781584888239-c20.

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

[22] 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.

[23] Tomas Kulik et al. "A Survey of Practical Formal Methods for Security". In: *Formal Aspects of Computing* (2022). Published Version. ISSN: 1433-299X. DOI: 10.1145/3522582. URL: https://eprints.whiterose.ac.uk/194011/.

[24] Edward A. Lee. "The past, present and future of cyber-physical systems: a focus on models". In: *Sensors (Basel)* 15.3 (Feb. 2015), pp. 4837–4869. DOI: 10.3390/s150304837.

[25] Abha Mahalwar and Rishabh Sharma. "Cyber-Physical Systems: Challenges and Future Directions". In: *Turkish Journal of Computer and Mathematics Education (TURCOMAT)* 11 (Dec. 2020), pp. 2865–2870. DOI: 10.61841/turcomat.v11i3.14651.

[26] Jan-Willem van de Meent et al. *An Introduction to Probabilistic Programming*. 2021. DOI: 10.48550/arXiv.1809.10756. URL: https://arxiv.org/abs/1809.10756.

[27] 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.

[28] A. Piedrafita and L. Barbini. "PHM Society European Conference". In: vol. 8. 1. June 2024, p. 7. DOI: 10.36001/phme.2024.v8i1.4055.

[29] Francesco Pontiggia, Ezio Bartocci, and Michele Chiari. *Model Checking Probabilistic Operator Precedence Automata*. 2025. arXiv: 2404.03515 [cs.LO]. URL: https://arxiv.org/abs/2404.03515.

[30] Francesco Pontiggia, Ezio Bartocci, and Michele Chiari. *POPACheck: a Model Checker for Probabilistic Pushdown Automata*. arXiv preprint arXiv:2502.03956. Conditionally accepted at CAV 2025. TU Wien, 2025.

[31] Tetsuya Sato et al. "Formal Verification of Higher-Order Probabilistic Programs: Reasoning about Approximation, Convergence, Bayesian Inference, and Optimization". In: *Proceedings of the ACM on Programming Languages* 3.POPL (2019), 38:1–38:30. DOI: 10.1145/3290351.

[32] 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.

[33] Sebastian Thrun. "Probabilistic robotics". In: *Communications of the ACM* 45.3 (Mar. 2002), pp. 52–57. DOI: 10.1145/504729.504754. URL: https://doi.org/10.1145/504729.504754.

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

[35] 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.

[36] Yizhou Zhang and Nada Amin. *Semantics and Contextual Equivalence for Probabilistic Programs with Nested Queries and Recursion*. Technical Report CS-2021-02. Yizhou Zhang (University of Waterloo); Nada Amin (Harvard University). Waterloo, Ontario, Canada: School of Computer Science, University of Waterloo, 2021.

Technological Implementation

[1] *ANTLR*. https://www.antlr.org/. Accessed 22 June 2025.

[2] *Chevrotain Documentation*. https://chevrotain.io/docs/. Accessed: 2025-07-15.

[3] *Chevrotain Parser*. https://chevrotain.io/docs/features/blazing_fast.html. Accessed 22 June 2025.

[4] *Extend or customize syntax highlighting*. GitHub Discussion #604, `eclipse-langium/langium`. Answered by msujew on July 19, 2022. 2022. URL: https://github.com/eclipse-langium/langium/discussions/604 (visited on 06/29/2025).

[5] *Jison*. https://zaach.github.io/jison/. Accessed 22 June 2025.

[6] *Langium Documentation: Document Lifecycle*. https://langium.org/docs/reference/document-lifecycle/?utm_source=chatgpt.com. Accessed: 2025-07-15.

[7] *Langium official website*. https://langium.org/. Accessed 24 June 2025.

[8] *Langium: A Framework for Language Engineering*. https://github.com/eclipse-langium/langium. Accessed 24 June 2025. 2025.

[9] *Language Server Extension Guide*. https://code.visualstudio.com/api/language-extensions/language-server-extension-guide. Accessed: 2025-07-15.

[10] *Language Server Protocol Specification*. https://microsoft.github.io/language-server-protocol/. Accessed 22 June 2025.

[11] *LSP Extension: Creating a extension*. https://learn.microsoft.com/en-us/visualstudio/extensibility/adding-an-lsp-extension. Accessed: 2025-07-15.

[12] *Nearley Parser*. https://nearley.js.org/. Accessed 22 June 2025.

[13] *PEG.js*. https://pegjs.org/. Accessed 22 June 2025.

[14] Mark Sujew. "Enabling unbounded lookahead in LL(k) parsers using adaptive LL(*): Performance analysis of the impact of adaptive LL(*) in the Chevrotain parsing library". Submitted on 21 April 2022. Master's thesis. Nordakademie, Elmshorn, Germany, Apr. 2022. URL: https://sujew.dev/assets/masterthesis.pdf?utm_source=chatgpt.com.

[15] *TextMate Language Grammars*. https://macromates.com/manual/en/language_grammars. Accessed 22 June 2025.

[16] *Tree-sitter*. https://tree-sitter.github.io/tree-sitter/. Accessed 22 June 2025.

[17] TypeScript Team. *Unions and Intersection Types*. Accessed 04 July 2025. Microsoft. 2025. URL: https://www.typescriptlang.org/docs/handbook/unions-and-intersections.html.

Abbreviations

API Application Programming Interface

AST Abstract Syntax Tree

BNF Backus-Naur Form

BOOL Boolean

CAV Computer-Aided Verification

CFG Context-Free Grammar

CLI Command-Line Interface

CPS Cyber-Physical System

CS Computer Science

CST Concrete Syntax Tree

DFA Deterministic Finite Automaton

DI Dependency Injection

DSL Domain-Specific Language

EBNF Extended Backus-Naur Form

ESM ECMAScript Module

JSON-RPC JavaScript Object Notation – Remote Procedure Call

LL Left-to-right, Leftmost derivation

LR Left-to-right, Rightmost derivation

LSP Language Server Protocol

LTL Linear Temporal Logic

MCMC Markov Chain Monte Carlo

OPA Operator Precedence Automaton

OPM Operator Precedence Matrix

PDA Pushdown Automaton

PEG Parsing Expression Grammar

POTL Procedence Operator Temporal Logic

PPL Probabilistic Programming Language

SMC Statistical Model Checking

SMT Satisfiability Modulo Theories

URI Uniform Resource Identifier

VI Variational Inference

VS Visual Studio

VSCE Visual Studio Code Extension

WS Web Services

XML eXtensible Markup Language

Program	*prog*	::=	*decl_list func func_list*
Declaration	*decl*	::=	*type identifier id_list*
		|	ϵ
	decl_list	::=	*decl* ; *decl_list*
		|	ϵ
	id_list	::=	, *identifier id_list*
		|	ϵ
Function	*func*	::=	*identifier* (*param_list*) { *decl_list block* }
FuncCall	*func_call*	::=	*identifier* (*arg_list*)
Argument	*arg*	::=	*expr*
	arg_list	::=	*arg arg_list_tail*
		|	ϵ
	arg_list_tail	::=	, *arg arg_list_tail*
		|	ϵ
Parameter	*param*	::=	*type amp_opt identifier*
Value-result	*amp_opt*	::=	&
		|	ϵ
	param_list	::=	*param param_list_tail*
		|	ϵ
	param_list_tail	::=	, *param param_list_tail*
		|	ϵ
Codeblock	*block*	::=	*stmt stmt_list*
	block_opt	::=	*block*
		|	ϵ
Statement	*stmt*	::=	*assignment* ;
		|	*query* ;
		|	*observation* ;
		|	*func_call* ;
		|	`throw` ;
		|	*if_then_else*
		|	*while*
		|	*try_catch*
	stmt_list	::=	*stmt stmt_list*
		|	ϵ
	assignment	::=	*lval* = *assign_rhs*
Right-hand side	*assign_rhs*	::=	*expr*
		|	*distribution*
Query	*query*	::=	`query` *func_call*
Conditioning	*observation*	::=	`observe` *expr*
IfThenElse	*if_then_else*	::=	`if` (*expr*) { *block_opt* } `else` { *block_opt* }
While-loop	*while*	::=	`while` (*expr*) { *block_opt* }
Try-catch	*try_catch*	::=	`try` { *block_opt* } `catch` { *block_opt* }

Left-hand value	*lval*	::=	*identifier* [*expr*]
		|	*identifier*
Sampling	*distribution*	::=	`Bernoulli` (*expr* , *expr*)
		|	`Uniform` (*expr* , *expr*)
Expression	*expr*	::=	*logical_or prob_assign_opt*
	prob_assign_opt	::=	*prob_assign*
		|	ϵ
Probabilistic Assignment	*prob_assign*	::=	*prob_choice expr*
	prob_choice	::=	{ *expr* `\:` *expr* }
	logical_or	::=	*logical_and logical_or_tail*
	logical_or_tail	::=	`||` *logical_and logical_or_tail*
		|	ϵ
	logical_and	::=	*comparison logical_and_tail*
	logical_and_tail	::=	`&&` *comparison logical_and_tail*
		|	ϵ
	comparison	::=	*term comparison_tail*
	comparison_tail	::=	*comparison_op term comparison_tail*
		|	ϵ
	comparison_op	::=	`>`
		|	`>=`
		|	`<`
		|	`<=`
		|	`==`
		|	`!=`
	term	::=	*factor term_tail*
	term_tail	::=	`+` *factor term_tail*
		|	`-` *factor term_tail*
		|	ϵ
	factor	::=	*division factor_tail*
	factor_tail	::=	`*` *division factor_tail*
		|	ϵ
	division	::=	*unary division_tail*
	division_tail	::=	`/` *unary division_tail*
		|	`%` *unary division_tail*
		|	ϵ
	unary	::=	*unary_op unary*
		|	*primary*
	unary_op	::=	`!`
		|	`-`
	primary	::=	*bool*
		|	*int_literal*
		|	*lval*
		|	(*expr*)

$$
\begin{array}{lcl}
\textit{identifier} & ::= & \textit{letter ident\_tail} \\
\textit{ident\_tail} & ::= & \textit{letter\_or\_digit ident\_tail} \\
 & | & \texttt{.}\ \textit{ident\_tail} \\
 & | & \texttt{:}\ \textit{ident\_tail} \\
 & | & \texttt{\~{}}\ \textit{ident\_tail} \\
 & | & \epsilon \\
\textit{letter} & ::= & \texttt{a} \\
 & | & \texttt{b} \\
 & | & \ldots \\
 & | & \texttt{Z} \\
 & | & \texttt{\_} \\
\textit{digit} & ::= & \texttt{0} \\
 & | & \ldots \\
 & | & \texttt{9} \\
\textit{letter\_or\_digit} & ::= & \textit{letter} \\
 & | & \textit{digit} \\
\textit{type} & ::= & \texttt{bool} \\
 & | & \textit{int\_spec} \\
 & | & \textit{int\_spec}\ [\ \textit{int\_literal}\ ] \\
\textit{digit\_seq} & ::= & \textit{digit digit\_seq} \\
 & | & \textit{digit} \\
\textit{int\_spec} & ::= & \texttt{u}\ \textit{digit\_seq} \\
 & | & \texttt{s}\ \textit{digit\_seq} \\
\textit{int\_literal} & ::= & \textit{digit\_seq int\_spec} \\
\textit{bool} & ::= & \texttt{true} \\
 & | & \texttt{false}
\end{array}
$$

Listing A.1: Backus-Naur form of MiniProb grammar

Expanded Code Listings

```
grammar MiniProb

type DeclOrParam = Decl | Param;

entry Program:
    (fileImports+=FileImport)*
    probabilisticQuery=PROB_QUERY? // not tested: out of scope
    formula=FORMULA? // not tested: out of scope
    'program:'?
    (declarations+=Decl ';')* functions+=Func+;

FileImport:
    '#include' file=STRING;

Decl:
    type=Type names+=ID (',' names+=ID)*;

Func:
    name=ID '(' params=ParamList? ')' '{'
        (declarations+=Decl ';')*
        body=Block
```

```
    '}';
FuncCall:
    ref=[Func:ID] '(' argumentList=ArgList? ')';

Block:
    statements+=Stmt+;

Stmt:
    ( Assignment | Query | Observation | FuncCall | 'throw') ';' | IfThenElse | While
    ↪  | TryCatch;

Lval:
    ref=[DeclOrParam:ID] ('['index=Expression']')?;

IfThenElse:
    'if' '(' condition=Expression ')' '{' thenBlock=Block? '}' 'else' '{'
    ↪  elseBlock=Block? '}';
While:
    'while' '(' condition=Expression ')' '{' whileBlock=Block? '}';
TryCatch:
    'try' '{' tryBlock=Block? '}' 'catch' '{' catchBlock=Block? '}';
Assignment:
    leftValue=Lval '=' (distribution=Distribution | expression=Expression);
Observation:
    'observe' condition=Expression;
Query:
    'query' function=FuncCall;

ParamList:
    parameters+=Param (',' parameters+=Param)*;
Param:
    type=Type byRef?='&'? name=ID;
ArgList:
    arguments+=Arg (',' arguments+=Arg)*;
Arg:
    expression=Expression; //only expre and no ref symbol
Distribution:
    (name='Bernoulli' '(' p=Expression ',' q=Expression ')') |
    (name='Uniform' '(' lower=Expression ',' upper=Expression ')');
ProbChoice:
    '{'numerator=Expression ':' denominator=Expression '}';

Expression:
    LogicalOr ({infer ProbabilisticAssignment.head=current}
    ↪  (probabilities+=(ProbChoice) fallbacks+=LogicalOr)+)?;
LogicalOr infers Expression:
    LogicalAnd ({infer BinaryExpression.left=current} operator='||'
    ↪  right=LogicalAnd)*;
LogicalAnd infers Expression:
    Comparison ({infer BinaryExpression.left=current} operator='&&'
    ↪  right=Comparison)*;
Comparison infers Expression:
    Term ({infer BinaryExpression.left=current} operator=('<' | '<=' | '>' | '>=' |
    ↪  '==' | '!=') right=Term)*;
Term infers Expression:
    Factor ({infer BinaryExpression.left=current} operator=('+' | '-') right=Factor)*;
Factor infers Expression:
    Division ({infer BinaryExpression.left=current} operator=('*') right=Division)*;
```

```
74  Division infers Expression:
75      Unary ({infer BinaryExpression.left=current} operator=('/' | '%') right=Unary)*;
76  Unary infers Expression:
77      ({infer LogicalNegation} operator='!') operand=Unary) | Primary;
78  Primary infers Expression:
79      {infer BoolLiteral} literal=BOOL |
80      {infer IntLiteral} literal=IntegerLiteral |
81      Lval |
82      '(' Expression ')';
83
84
85  Type:
86      'bool' | IntType;
87  IntType:
88    prefix=INT_PREFIX ({infer IntArray} '[' size=INT ']')?;
89  IntegerLiteral: // same name as infer IntLiteral creates minor conflicts
90      sign=('+' | '-')? value=INT suffix=INT_PREFIX;
91
92  terminal INT_PREFIX: /[su][1-9][0-9]{0,8}/; // 2^29-1 backend
93  terminal BOOL returns boolean: /(true|false)/;
94  terminal PROB_QUERY: /probabilistic query:.*;/;
95  terminal FORMULA: /formula\s?=\s?.*;/;
96  terminal ID: /(?!(true|false)|[su][1-9][0-9]*)[a-zA-Z_][a-zA-Z0-9_\.\:\~]*/;
97  terminal STRING returns string: /"(\\.|[^"\\])*"|'(\\.|[^'\\])*'/;
98  terminal INT returns number: /[0-9]+/;
99  hidden terminal WS: /\s+/;
100 hidden terminal ML_COMMENT: /\/\*[\s\S]*?\*\//;
101 hidden terminal SL_COMMENT: /\/\/[^\n\r]*/;
```

Listing B.1: Langium grammar definition of MiniProb (LL(2) parsable)