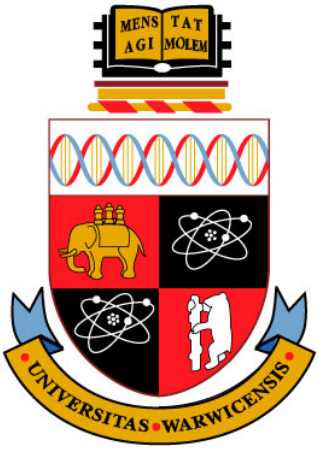

YAPPL: A Probabilistic Programming Language

Third Year Dissertation

Edward Denton (5515605)

Supervisor: Alex Dixon

Year of Study: 2025/2026

Department of Computer Science

University of Warwick

Abstract

There are problems in computer science with no known deterministic polynomial-time algorithms but which can be solved efficiently by a probabilistic Turing machine, characterised by complexity classes such as RP, coRP, and BPP. Yet the vast majority of mainstream programming languages are not built for probabilistic programming: distributions, confidence levels, and randomised error bounds must be tracked manually by the programmer, and the bookkeeping is error-prone. This project addresses that gap by providing a toolkit of language features designed to make probabilistic programming easier.

The result is YAPPL (Yet Another Probabilistic Programming Language), a standalone language combining standard programming features - functions, control flow, user-defined enumerations - with first-class distributions, distribution-level operations, and probabilistic functions annotated with their RP, coRP, or BPP error class. The runtime amplifies confidence automatically by running the appropriate number of rounds for a programmer-specified target, and supports analytical and Bayesian extraction of underlying distributions. These features compose: typed Markov chains over user-defined enumerations are presented as one example of the more complex programs that can be expressed concisely in the language. The interpreter is implemented as a tree-walker in Rust, and its testing strategy and deployment as a public web playground are discussed.

Keywords: Probabilistic Programming, Programming Language Design, Randomised Algorithms, RP, coRP, BPP, Markov Chains, Rust.

Acknowledgements

My deepest thanks go to my supervisor, Alex Dixon, whose patience and guidance shaped this project from the very first meeting. Alex helped flesh out the core ideas of the language, offered invaluable advice on managing and presenting the work, and gave detailed feedback on both the project specification and the progress report. I hope the finished project proves worthy of all that time and care.

I am also grateful to my close friends James Lai Gray and Lucy Siket, who have stuck by me since first year and whose constant support has made my time at University genuinely enjoyable. In the same spirit, I would like to thank Catherine Hibbs for making sure I stepped away from work every now and then; those 'stress-relief' breaks meant more than they may have seemed at the time.

Finally, this project would not have been possible without the long line of researchers and engineers whose work in probabilistic programming I have built on.

Declarations

A large language model (Claude) was used to assist with generating test cases for the unit test suite; every generated test was reviewed and verified by hand before being committed.

Some portions of this report have been adapted or reused from material submitted earlier in the project specification and the progress report.

Links

The web playground should be live at: `https://yappl.containers.uwcs.co.uk/` The source code can be found at: `https://github.com/RexMortem/3YP`

Contents

List of Figures

List of Tables

1 Introduction

1.1 Motivation

Randomness is often viewed as something a careful programmer should try to avoid. Unpredictable behaviour is generally undesirable in software, and a substantial body of programming-language research has been devoted to making programs easier to reason about by reducing the amount of non-determinism they exhibit. Functional programming, for instance, treats referential transparency as a core property precisely because it makes programs easier to predict.

Despite this, randomness is among the most useful tools in modern algorithm design. Permitting an algorithm to make random choices often allows it to avoid worst-case behaviour, simplify its structure, or achieve better expected performance than is possible with any deterministic approach [20, 19]. Many of the algorithms that underpin modern infrastructure rely on this trade-off: randomised quicksort, universal hashing, primality testing for cryptography [27, 23], and sketch-based summaries for data streams such as Bloom filters [6] and the Count-Min sketch [9] all give probabilistic guarantees rather than deterministic ones, and in return they are simpler, faster, or more memory-efficient than their deterministic counterparts.

The theoretical case for randomness is just as strong. There are decision problems for which efficient probabilistic algorithms exist (formally, problems in the complexity class **RP**) but for which no efficient deterministic algorithm is known [26]. Whether this gap is real, or whether every problem in RP can in principle be solved deterministically with comparable efficiency, is one of the long-standing open questions in theoretical computer science. The trade-off between correctness and performance can typically be tuned by varying the number of independent rounds: more rounds yield a higher probability of a correct answer, at a proportional cost in time [20].

Given how central probability has become to algorithm design, it would be useful for a programming language to treat probabilistic concerns as a first-class concern of the language itself, rather than as bookkeeping that the programmer must perform by hand. In particular, such a language should provide:

- mechanisms for analysing the confidence and behaviour of probabilistic algorithms;

- the ability to model and manipulate underlying probability distributions as first-class values;
- the ability to combine distributions and probabilistic algorithms compositionally.

This project, YAPPL (Yet Another Probabilistic Programming Language), sets out to design and implement such a language, with a focus on ergonomic features for the construction and analysis of randomised algorithms. The original problem statement [11] (reproduced in Appendix B) positioned YAPPL as a contribution to probabilistic programming with applications in areas as varied as predictive modelling, intrusion detection, and machine learning. YAPPL is necessarily a prototype rather than a production tool, since a fully production-ready compiler is well beyond the scope of a third-year project. The aim is instead to investigate which language features actually improve the experience of writing and reasoning about randomised algorithms, and to evaluate the design choices that distinguish a probabilistic-first language from a general-purpose one.

1.2 Theoretical Background

The background reading for this project covers the theory of randomised algorithms, the design space of probabilistic programming systems, and the existing tools and languages that occupy that space. This subsection summarises the most influential of them.

1.2.1 Randomised algorithms

Randomised algorithms are conventionally divided into two families [18, 20]:

- **Monte Carlo** algorithms always run within a stated time budget but may return an incorrect answer with bounded probability.
- **Las Vegas** algorithms always return a correct answer, but their running time is itself a random variable.

Both families are widely used, but YAPPL focuses on the Monte Carlo case. Monte Carlo algorithms are the more interesting of the two from a programming-language perspective: the programmer has to reason explicitly about confidence levels and error rates, and a language that hides this reasoning would deprive the programmer of exactly the feedback they need.

Within the Monte Carlo world, three complexity classes provide a fine-grained vocabulary for the kind of error an algorithm can make [26, 19]:

- **RP** (Randomised Polynomial-time): one-sided error. An RP algorithm may incorrectly accept (a false positive: it says "yes" when the answer is "no") but never incorrectly rejects. A "no" answer is therefore guaranteed to be correct; a "yes" answer is only probably correct.
- **coRP**: the symmetric counterpart of RP, with the roles of "yes" and "no" swapped. A coRP algorithm may incorrectly reject (a false negative) but never incorrectly accepts; a "yes" answer is therefore guaranteed to be correct, while a "no" answer is only probably correct. Solovay-Strassen primality testing is RP (a non-prime witness is conclusive); Fermat compositeness testing is coRP (a Fermat witness is conclusive).
- **BPP** (Bounded-error Probabilistic Polynomial-time): two-sided error. Both directions of mistake are possible, but each is bounded below 1/2, so confidence can be amplified by a majority vote across independent rounds.

These three error classes guide the design of YAPPL's probabilistic functions and motivate the language's confidence-based round amplification (calculating how many rounds to run for a given confidence level), discussed in detail in Chapter 2.

Concrete examples of such algorithms are not hard to find. The Solovay–Strassen [27] and Miller–Rabin [23] primality tests are RP algorithms whose accepting answers are uncertain but whose rejecting answers are conclusive. The Schwartz–Zippel lemma [24] underlies polynomial identity testing, a coRP procedure that is still the most efficient way to verify large algebraic identities. Sketching techniques such as Bloom filters [6] and the Count-Min sketch [9] use random hashing to obtain compact data summaries with controllable error rates. Each of these is straightforward to express on paper but tedious to express in a general-purpose language, where the bookkeeping of probabilities, witnesses, and confidence accumulation is manual and error-prone.

1.2.2 Existing tools and languages

A number of existing libraries and languages occupy parts of the probabilistic programming space, and YAPPL takes inspiration from several of them while deliberately departing from others.

Data.Distribution (Haskell). Haskell's `Data.Distribution` is an embedded library that represents discrete distributions as first-class values. Distributions can be sampled, reweighted, and composed monadically; the documentation on Hackage [1] sets out the full algebra. Two distributions can be combined into a third whose support is the Cartesian product of the original supports, which makes operations such as "sum of two dice" a one-line expression. This compositional style directly inspired the distribution combinators in YAPPL (discussed in Chapter 2), although YAPPL extends the idea to a richer set of distribution types and to typed Markov chains over user-defined enumerations.

R2 (Microsoft Research). R2 is a probabilistic programming system whose front-end is a small imperative language and whose back-end performs program transformations to enable efficient Markov chain Monte Carlo inference [21]. The R2 paper demonstrates that quite a simple host language is sufficient to express interesting probabilistic models, and that a tractable inference engine can be built around it. R2's emphasis on transformations as a route to efficiency is something YAPPL does not pursue (the YAPPL interpreter is direct and tree-walking) but the broader observation that a small language can still capture meaningful probabilistic computations is one this project adopts.

Pyro and ProbTorch. Pyro [5] and ProbTorch [25] are deep probabilistic programming libraries built on top of PyTorch, primarily designed for variational and amortised inference over neural network models. Bingham et al. describe Pyro as targeting "deep universal probabilistic programming" [5]: it is powerful but heavyweight, and trades simplicity for expressivity over continuous, high-dimensional models. These systems target a different audience from YAPPL. Their target user is the machine learning practitioner building generative models, whereas YAPPL is concerned with the more focused setting of randomised algorithms and their analysis. Studying these systems was useful primarily as a contrast, and as a reminder that "probabilistic programming language" covers several quite different design goals.

Prolog. Prolog was investigated as a logic language whose primitives might lend themselves to probabilistic semantics. Its central concepts are facts, rules, and queries, and the resolution procedure produces the set of values that satisfy a given query, when such a set exists [8]. Selecting one of these values, or enumerating them all, corresponds to what a probabilistic programming language might do at a sample point or a marginalisation point. Prolog's resolution model was not adopted as a basis for YAPPL: enumerating

satisfying values is not by itself sufficient for writing Monte Carlo algorithms within the available time, and the conceptual gap between Prolog's resolution model and the imperative style typical of randomised algorithm pseudocode would have been a substantial barrier to readability.

Verse. Verse is a recent functional-logic language from Epic Games, designed for the eventual scripting of large interactive systems. Its core calculus, formalised by Augustsson et al. [3], is notable for its declarative treatment of choice and constraint. Verse demonstrates that non-deterministic choice can be incorporated into a modern language without becoming unwieldy. YAPPL does not adopt Verse's functional-logic substrate, but the underlying principle that probabilistic and non-deterministic features should be integrated rather than added as an afterthought is shared with this project.

A common observation across these systems is that the most usable probabilistic abstractions are those that treat distributions as ordinary values and manipulate them with a small, principled algebra. YAPPL adopts this as its basis and adds explicit, first-class machinery for analysing the error behaviour of randomised algorithms, an area in which the existing systems offer relatively little support compared to their inference machinery for continuous models.

1.3 Methodology

Three substantial methodological choices shape this project: the decision to design a standalone language rather than an embedded Domain-Specific Language (DSL); the decision to build an interpreter rather than a compiler; and the decision to write the implementation in Rust.

Standalone language vs. embedded DSL. An embedded DSL would have been considerably easier to construct. A host language such as Haskell or Rust already provides a parser, a type system, a runtime, and a library ecosystem, and an embedded DSL inherits all of these for free. The cost of that convenience is loss of control: the semantics of an embedded language are constrained by the semantics of its host, error reporting goes through the host's compiler, and surface syntax is limited by what the host's overloading and macro systems allow. A standalone language, by contrast, can choose its own semantics for sampling, distribution composition, and confidence propagation; can present error messages tailored to probabilistic concepts; and can offer a coherent surface grammar

without being constrained by the host. Since the explicit goal of this project is to evaluate language features for probabilistic programming, building a standalone language was the more appropriate choice.

Interpreter vs. compiler. A compiler would yield superior runtime performance and would in principle scale to larger inputs [2], and an interpreter for YAPPL is not going to compete with hand-tuned C on serious workloads. However, the focus of this project is on the language's features and ergonomics, not on the throughput of any particular benchmark. Implementing a code-generating back-end, register allocation, and an optimising middle-end would have consumed a disproportionate share of the available time and detracted from the questions the project actually exists to answer. An interpreter, by contrast, supports very rapid prototyping: a new language feature can be added, evaluated, and either kept or discarded within a single sitting. This proved important when iterating on features such as confidence-amplified probabilistic functions and the `bind` and `step` operators for Markov chains, both of which went through several design revisions before settling.

Rust as the implementation language. Rust was chosen for two reasons. The first is its safety: Rust's ownership and borrowing model catches an entire class of bugs at compile time that would otherwise need to be hunted at runtime, which makes a relatively large project tractable for a single developer. Recent benchmarking work by Ivanov [17] also shows that Rust's runtime performance is broadly competitive with C++ on routine workloads, so the safety guarantees do not come at a meaningful performance cost for an interpreter of this size. The second reason is the existence of `nom` [10], a parser combinator library that allows recursive descent parsers to be written almost directly from a BNF grammar. This kept the cost of changing the surface syntax of YAPPL very low, which proved valuable as the language grew distribution literals, method calls, enum types, and Markov chain primitives over the course of the year.

1.4 Implementation Background

YAPPL follows the conventional structure of an interpreted language. Source text is fed into a parser, which produces an abstract syntax tree (AST). The tree is then walked by an interpreter that maintains an environment of values, function definitions, and (in the final language) enum and distribution information. Each phase corresponds to a clearly separated module in the codebase, in line

with normal separation-of-concerns hygiene.

The parser is built using `nom` [10], which represents parsers as composable Rust functions of type `&str -> IResult<&str, T>`. Combinators such as `alt`, `many0`, `delimited`, and `terminated` compose smaller parsers into larger ones in a way that closely mirrors the BNF productions of the grammar. The benefit of this approach is that the parser reads almost as a transliteration of the grammar; the cost is that some standard parsing techniques (such as direct left recursion) need to be expressed slightly differently, since combinator parsers are essentially recursive descent and would loop forever on a left-recursive production.

Two principal styles of parsing are described in the standard reference [16]: top-down parsers begin at the start symbol of the grammar and ask how the input could have been derived, while bottom-up parsers begin at the leaves and ask which productions could have produced them. Bottom-up parsers (such as LR parsers) are generally more powerful and tend to produce smaller parser tables for large languages, but they are considerably more complex to implement and to debug. For a small, evolving language such as YAPPL, where a more sophisticated parser would not be a productive use of the limited time available for a third-year project, the simplicity and rapid iterability of recursive descent outweigh its theoretical limitations, and the cost of backtracking is negligible because programs are short and the grammar is shallow. This project therefore uses top-down parsing in the form of recursive descent with backtracking.

After parsing, the resulting AST is consumed by a tree-walking interpreter that performs a depth-first traversal, evaluating leaf nodes first and propagating values upwards. The interpreter maintains an environment of bound variables and registered (regular and probabilistic) functions, and produces an output stream of text and visualisation records that can be rendered either as ASCII histograms in the terminal or as inline SVG in the web playground. This style of interpreter is straightforward to reason about and to extend, which has been important as the language has grown over the course of the project. The remaining chapters describe the resulting language, its features, and the lessons learned in building it.

2 The YAPPL Language

This chapter describes YAPPL in terms of the principles that guided its design, the features that distinguish it from a general-purpose language, and the kinds of programs that can be expressed in it. A complete grammar and reference for the language is given in Appendix A; the present chapter takes the perspective of a programmer who wants to know why the language looks the way it does, and reproduces only those snippets of the specification that are most informative.

2.1 Principles

The full design of YAPPL is set out as five design pillars in Section 1.2 of the language specification (Appendix A). This subsection summarises those pillars in plain prose, in the same order they are presented in the specification, so that the rest of the chapter can be read against the same vocabulary.

- **(a) Distributions are values, and the language works on them for as long as it can.** A probabilistic program in YAPPL talks about `uniform(1,6)` or `Bernoulli(0.3)` as ordinary first-class values, combines them with arithmetic and method calls, and only converts to a concrete sample (or to an exact probability) at the very end. This is the same idea that makes Haskell's `Data.Distribution` convenient to use [1], generalised to a wider set of distribution constructors and to typed transitions over user-defined enums. The motivating principle is to obtain concrete values as late as possible.
- **(b) Probabilistic functions declare their error in their signature.** The complexity classes RP, coRP, and BPP draw a sharp distinction between which side of an answer is reliable [26]; YAPPL puts that distinction directly into function signatures via the `pb function` construct, so that the error behaviour of a randomised algorithm is part of its type rather than something the programmer has to track in comments.
- **(c) The compiler derives the repetition strategy from the error class.** Once a function declares its error class, the runtime knows how to amplify a single round into a confident answer: short-circuiting on `Certain` for RP and coRP, majority vote for BPP. The user writes the single-round body and the language handles the loop, the round count, and the confidence accumulation.

- **(d) Confidence targets are the user-facing abstraction.** The caller of a probabilistic function specifies how confident they want to be, not how many rounds to run. The runtime translates the target into the smallest sufficient number of rounds and reports both the chosen round count and the actual confidence achieved alongside the result. The same mechanism distributes a shared confidence budget across a batch of probabilistic calls via the union (Bonferroni) bound, so that programmers do not have to perform the union bound by hand.
- **(e) Distributions and probabilistic functions are dual views of the same object, as much as possible.** Every probabilistic function has an implicit per-round distribution, and YAPPL makes this explicit through the `distribution_of` operator, which reifies that distribution as a first-class value. The reverse direction, treating a distribution as the input to a transition function, is supported by `bind` and `step`. Neither direction is a perfect equivalence, but together they let programmers move between the two views wherever it is useful to do so.

A sixth, less formal principle is that the language should be small. Every feature in YAPPL is justified by a category of program that the project actually wanted to write; nothing has been added because a more mature language would have it. Where a feature would have duplicated host-language machinery without adding value specific to probabilistic programming (a module system, for example), it has been left out.

2.2 Primary features

2.2.1 Distributions as first-class values

The set of built-in distribution constructors is small but the relationships between them are not obvious at first glance. Figure 1 sketches the special-case relationships: `Bernoulli` is the one-trial special case of `Binomial`, `Geometric` answers the dual "how many trials until the first success" question for the same underlying coin, `uniform` is a particular shape of `Discrete` (a flat probability mass over a contiguous integer range), and `Discrete` is the most general finite distribution constructor in the language. The two continuous constructors (`uniformContinuous` and `Beta`) sit on a separate branch since the language treats them through sampling rather than analytical enumeration.

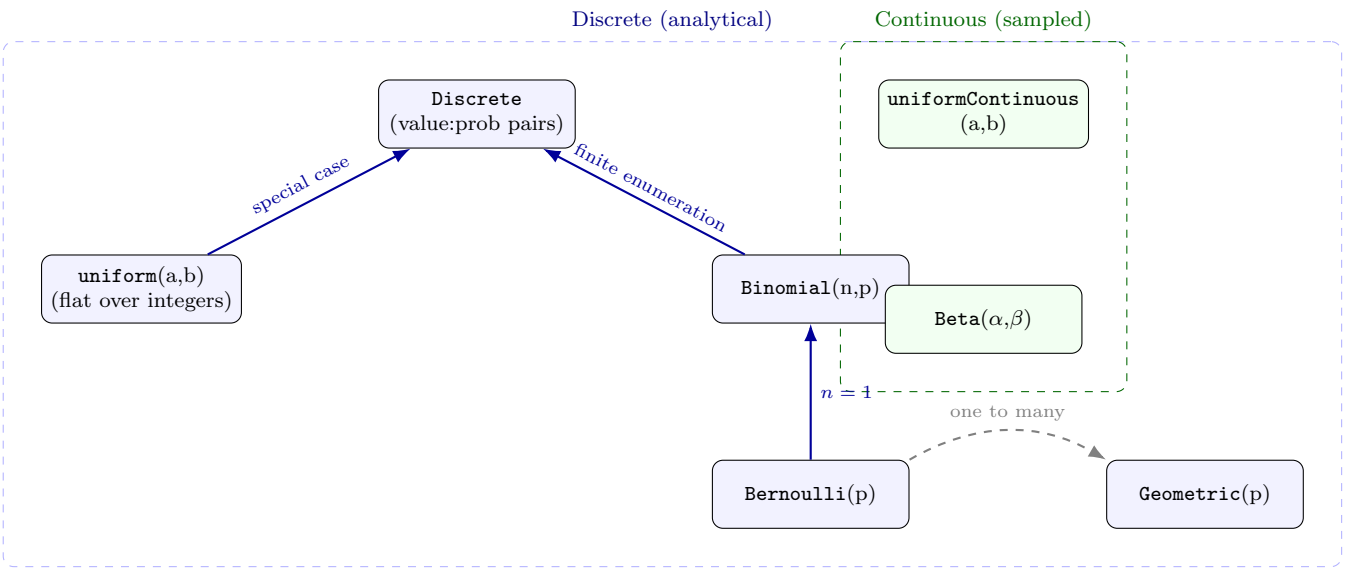

Figure 1: Special-case relationships between the built-in distribution constructors in YAPPL. Solid arrows point from a special case to its parent (`Bernoulli`(p) =`Binomial`$(1, p)$; `uniform`(a, b) is a flat instance of `Discrete`; `Binomial` is itself representable as a finite `Discrete` over $\{0, \ldots, n\}$). The dashed "one to many" arrow points from `Bernoulli` to `Geometric`: the same coin can be lifted from a single trial to a stream of trials by asking how many flips it takes for the first success to occur. The two continuous constructors live on a separate branch because YAPPL handles them through sampling rather than analytical enumeration.

The most basic move in YAPPL is to treat a distribution as an ordinary value. The constructors `uniform`, `uniformContinuous`, `Bernoulli`, `Binomial`, `Geometric`, and the user-facing `Discrete` produce values that can be bound to variables, passed to functions, returned from functions, combined arithmetically, and queried with method calls. The combinators that act on these values (in particular the lifted `+` operator for convolution, and the `bind` and `step` operators for Markov chains) were directly inspired by Haskell and the `Data.Distribution` library introduced in Chapter 1 [1]; YAPPL takes the same monadic style of combination and applies it to a wider set of distribution constructors, and to typed transitions over user-defined enums.

```
let die = uniform(1, 6);
let avg  = die.mean();       // 3.5
let p5   = die.expect(5);    // 1/6 (exact Fraction)
let roll = die.sample();     // a random integer in [1,6]
```

Listing 1: Querying a uniform distribution

The available method set is small and deliberately so: `sample`, `expect`, `mean`, `visualise`, and (for uniform distributions) `min` and `max`. Each of these answers a question that a programmer reasoning about a randomised algorithm actually asks: what a single draw looks like, the exact probability of a particular outcome, the expected value, and the shape of the distribution.

A consequence of treating distributions as values is that arithmetic on them is meaningful. Adding two distributions does not sample from them and add the samples; it produces a new distribution over all possible sums, computed analytically when both operands are discrete [19]:

```
let die = uniform(1, 6);
let twoRolls = die + die;
output(twoRolls.expect(7));   // 6/36, the modal sum
output(twoRolls.expect(12));  // 1/36, the rarest
```

Listing 2: Adding two dice analytically

The same code in a library-based system would either sample many times and approximate `expect(7)`, or require the programmer to write the convolution by hand. In YAPPL it is one line, and the answer is exact, and after the `+` operator was lifted to distributions several of the test programs reduced from a dozen lines to single expressions.

Limits of analytical arithmetic. The restricted choice of operator addresses a limitation raised in feedback on the progress report: the family of standard distributions is not closed under arithmetic. Adding two integer uniforms gives a discrete triangular distribution that can still be enumerated exactly, but multiplying two continuous uniforms gives a density involving $-\log u$ over $(0, 1]$, and once division and mixtures enter the picture the analytical forms stop fitting into any finite catalogue of constructors. Tracking every such product or quotient as a closed-form value is out of scope for a third-year project.

YAPPL therefore restricts its analytical machinery to the cases where it is cheap and exact. The `+` operator is lifted to distributions only when both operands are discrete, where the result is a finite enumeration of (sum, probability) pairs and the support stays bounded; the computation is the convolution $(D_1 \oplus D_2)(k) = \sum_{i+j=k} D_1(i)D_2(j)$. For continuous distributions, products, and quotients, the programmer draws a `sample` from each operand and combines them in the ordinary numeric world, or approximates the result by Monte Carlo sampling.

This is a real limitation, but a manageable one. The algorithms YAPPL was designed to support (Solovay-Strassen, Fermat compositeness, BPP bias detection, enum Markov chains) are predominantly discrete, and the analytical operators are appropriate for them. The continuous case can still be modelled operationally by sampling, while keeping the distribution first-class up to the sample point. A future revision could extend the analytical machinery to more continuous distributions through a hybrid mode that handles closed-form combinations where possible and falls back to Monte Carlo sampling otherwise; this is discussed in Chapter 4.

Was this ergonomic? Within the limits described above, yes. Analytical arithmetic combined with `expect` made test programs about dice, coins, and small discrete experiments significantly shorter than their general-purpose equivalents, and because `expect` returns an exact `Fraction`, the result can be checked against a hand calculation without floating-point error.

2.2.2 Probabilistic functions and error classes

The defining feature of YAPPL is the `pb function` construct. A `pb function` declares its error class up front (one of `RP`, `coRP`, or `BPP`) and inside its body returns either `Certain(x)` (a definitive answer that the runtime should immediately accept) or `Uncertain(x)` (a probabilistic vote that the runtime should weigh against other rounds). The caller never runs a `pb function` directly; instead they invoke it with a confidence target:

```
pb function is_prime(p: int) -> bool {
    error_class: RP,
    error_distribution: Geometric
} {
    if p < 2 { return Certain(false); };
    if p == 2 { return Certain(true); };
    if p % 2 == 0 { return Certain(false); };

    a = uniform(1, p - 1).sample();
    jacobian = (p + jacobi(a, p)) % p;
    euler = mod_exp(a, (p - 1) / 2, p);

    if jacobian == 0      { return Certain(false); };
    if euler != jacobian  { return Certain(false); };
    return Uncertain(true);
}

let result, info = is_prime(53) with confidence >= 0.9;
```

Listing 3: Solovay-Strassen primality, abbreviated

The runtime reads the error class and the supplied confidence target, computes the number of rounds needed to amplify the per-round error budget below the requested level, and runs that many rounds. For RP and coRP this is the geometric calculation $k = \lceil -\log_2(1-c) \rceil$ [27]. For BPP the runtime uses a *Chernoff bound*: a standard tail bound that quantifies how unlikely it is for the empirical fraction of correct rounds in k independent trials to deviate substantially from its true mean, and that decays exponentially in k [20, 19]. Under the standard BPP assumption that each round has success probability 3/4, applying that bound to the majority vote across k rounds gives an error of at most $e^{-k/8}$, so the smallest sufficient round count for a target c is $k = \lceil -8\ln(1-c) \rceil$. The second binding (`info`) carries the actual round count and the resulting confidence, so the caller can audit what the runtime did.

The point of the feature. The shape of the error class is the language's responsibility, while the per-round logic is the programmer's. A pb function written to be RP can be reused at any confidence level without rewriting any of its body; the runtime alone is responsible for choosing the round count, and the runtime alone has to be debugged when it gets the arithmetic wrong. This separation matched the project's original goal of putting confidence-based reasoning into the language rather than leaving it as a comment for the programmer to maintain.

Was this ergonomic? Yes, but with caveats. The body of `is_prime` above is identical to the body that a non-probabilistic language would have, plus a single `Certain`/`Uncertain` marker on each return, with no manual loop, no manual round count, and no manual confidence accumulation. The principal cost was that the metadata block (`error_class` and `error_distribution`) is verbose for trivial functions, and that BPP's two-sided amplification is conceptually heavier than the RP/coRP case: programmers have to internalise the difference between a guaranteed-correct answer on one side and a majority vote that is only probably correct. Both observations are revisited in Chapter 4.

2.2.3 `distribution_of`: introspecting a probabilistic function

Every `pb function` has an implicit per-round distribution: the probability that any given round returns `Certain` versus `Uncertain`. YAPPL exposes this as a first-class object via `distribution_of`, with three modes:

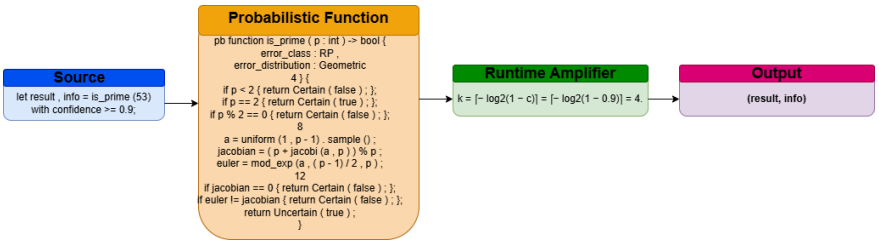

Figure 2: Calling a `pb function` in YAPPL. The programmer writes a single-round body that returns either `Certain(v)` or `Uncertain(v)` and tags the function with an error class (`RP`, `coRP`, or `BPP`); at the call site, the programmer specifies a confidence target c. The runtime then performs round amplification, which means computing how many independent rounds k are needed for the chosen error class to bring the overall error below $1 - c$, running that many rounds, and binding the result together with an `Info` record reporting the actual round count and the achieved confidence. For RP and coRP the formula is $k = \lceil -\log_2(1-c) \rceil$; for BPP it is $k = \lceil -8 \ln(1-c) \rceil$.

```
1 let analytic  = distribution_of(is_prime(53), analytical);
2 let empirical = distribution_of(is_prime(53), empirical,
    200);
3 let bayesian  = distribution_of(is_prime(53), bayesian,
    200);
```

The `analytical` mode returns the worst-case distribution implied by the declared error class. The `empirical` mode runs the function N times and reports the observed Certain rate as a Bernoulli distribution. The `bayesian` mode runs N rounds and returns the Beta posterior over the per-round Certain probability, starting from a uniform prior; the Beta distribution is the conjugate prior for a Bernoulli rate, so the posterior is itself a Beta whose parameters are the prior parameters incremented by the observed counts [14]. All three modes return ordinary distribution values that can be visualised, sampled from, or asked for their mean.

This feature does not shorten any single program, but it enables interactive exploration of a probabilistic function's per-round behaviour. The Bayesian view was useful when debugging the Solovay-Strassen implementation: the posterior over the Certain rate showed at a glance whether witness selection was finding compositeness as often as theory predicted, in cases where the analytical bound and the empirical observation disagreed.

Was this ergonomic? Mostly. The three modes give a progression from cheap theoretical bounds to more expensive empirical estimates, and unifying them under a single distribution type kept the

call sites uniform. The principal awkwardness is that the syntax takes a bare function call expression rather than a value, making it a special form; this is an unavoidable consequence of needing to re-run the function and is documented in the spec.

2.2.4 `map` with a shared confidence budget

Applying a probabilistic function to a list of inputs raises an immediate question: how should the confidence budget be split? YAPPL's answer is to make this explicit at the call site. The `map` construct accepts an optional `with confidence >= c` clause, and when it is present the runtime applies the union bound (also known as the Bonferroni inequality [19, 7]): the per-element confidence is $1-(1-c)/n$, so that the probability of any element being wrong is at most $1-c$.

```
let primes = map(is_prime, [2,3,4,5,6,7,8,9,10]) with
    confidence >= 0.99;
// -> [true, true, false, true, false, true, false, false,
    false]
```

Listing 4: Confidence-aware map

When applied to a regular function the `with confidence` clause is omitted and `map` behaves like the ordinary higher-order function.

Was this ergonomic? This was the feature whose payoff I underestimated most. The union-bound calculation is easy to perform incorrectly by hand, since it is tempting to apply an overall budget to each element individually and end up with a much weaker guarantee than intended. Bundling the calculation into a single keyword removed an entire category of subtle bugs from the test programs.

2.2.5 Markov chains over enums: `bind` and `step`

YAPPL has user-defined enums and typed discrete distributions over them (`Discrete<Weather>`). On top of these, the `bind` and `step` primitives express one-step and n-step Markov chain evolution [22]. `bind(d, f)` threads a transition function $f : T \to$ `Discrete<T>` over a current distribution d, producing the next-step distribution; `step(s, f, n)` starts from a pure state s and applies f for n steps.

```
enum Weather { Cloudy, Rainy, Sunny }

fn weather_transition(today: Weather) -> Discrete<Weather> {
    if today == Sunny  { return Discrete(Sunny: 0.7, Cloudy:
        0.2, Rainy: 0.1); };
    if today == Cloudy { return Discrete(Cloudy: 0.7, Sunny:
        0.2, Rainy: 0.1); };
```

```
    return Discrete(Rainy: 0.7, Cloudy: 0.2, Sunny: 0.1);
}

let after_50 = step(Sunny, weather_transition, 50);
output(after_50.visualise());   // close to the stationary
    distribution
```

Listing 5: A weather Markov chain in YAPPL.

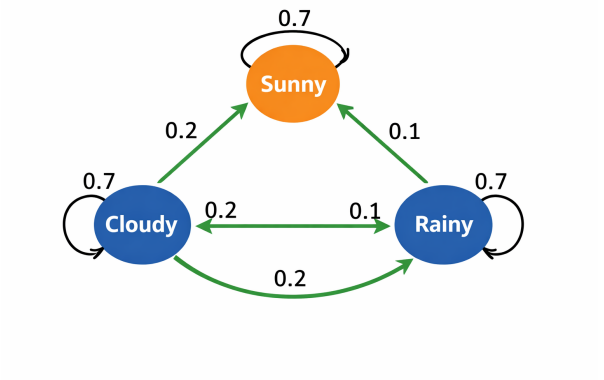

Figure 3: A state-transition diagram visualisation of the weather Markov chain defined by `weather_transition` above. Each node is a state (`Sunny`, `Cloudy`, or `Rainy`) and each arrow is annotated with the per-step probability of moving from one state to another. `bind` threads this transition function over a current distribution to produce the next-step distribution, and `step` iterates that operation n times until the distribution is close to its stationary fixed point.

Typing rules ensure that the variant names in a `Discrete` literal must belong to the declared enum, catching the most common transcription error before the program ever runs.

Was this ergonomic? Enums, typed discrete distributions, and `bind`/`step` became one of the most effective parts of the language. Writing a Markov chain in a general-purpose language requires the programmer to invent a state encoding and write the transition matrix multiplication by hand; in YAPPL the state space is named, the transition is a function that returns a distribution, and the chain evolution is a single call. The cost was the implementation effort of the type-checker for typed distributions, which is discussed in Chapter 3.

2.2.6 Approximate equality

A small but useful feature is the `~=` operator with an optional `within` clause: `x ~= y within 0.01` returns true when `x` and `y` differ by at

most 0.01. This exists because the questions I asked about probabilistic computations almost always involved floating-point comparisons, and `==` on floats is a known source of subtle bugs [15]. The feature is small, but it removed enough boilerplate from the test suite to justify its inclusion.

2.3 Programs in the language

The features described above compose. This subsection sketches three slightly larger programs that put several of them together; the full source for each lives under `Sample/Probabilistic/` in the project repository.

Bias detection (BPP). Suppose we are given a coin of unknown bias p and want to decide whether $p \geq 0.5$. A single sample is wrong with probability $\min(p, 1-p)$, so this is a two-sided BPP problem with no zero-error direction. The pb function is one Bernoulli sample and a single `if`; the runtime handles the majority vote and the Chernoff round count [20]. With a target confidence of 0.99 and $p = 0.75$ the runtime selects 37 rounds and almost always returns the correct answer; the caller never has to know that 37 was derived from $\lceil -8 \ln(0.01) \rceil$.

Vetting a list of candidate primes (RP + map). The `is_prime` pb function from the previous section is composed with confidence-aware `map` to vet a whole list of integers in one expression. The union bound makes the cost of each element grow only logarithmically with the list length (because the per-element budget is $1-(1-c)/n$ and the round count is logarithmic in that), so vetting a list of nine inputs at overall confidence 0.99 requires only ten rounds per element. The same program written in a general-purpose language would have at least three nested concerns the programmer would have to keep straight: the per-round Solovay-Strassen logic, the per-element confidence amplification, and the union-bound split. In YAPPL each of these lives in exactly one place.

Stationary distribution of a weather chain (enums + step + visualise). The Markov chain example from earlier combines the language's enum machinery with its visualisation: `step(Sunny, weather_transition, 50).visualise()` runs the chain to near-stationary [22] and renders the resulting distribution either as an ASCII histogram in the terminal or as inline SVG in the web playground. There is no explicit

matrix anywhere in the program; the chain is described entirely by its transition function.

For comparison, Listing 6 shows the same weather Markov chain written in plain Python. The states have to be encoded as integer indices, the transition function has to be flattened into a 3×3 matrix, and the chain evolution has to be expressed as a hand-written loop of matrix-vector multiplications. Contrast this against Listing 5 (the YAPPL version above): the YAPPL program names the state space directly with an `enum`, expresses the transition as an ordinary function returning a `Discrete`, and collapses the chain evolution to a single `step` call, with no explicit matrix anywhere in the source.

```
# States must be encoded as integer indices.
SUNNY, CLOUDY, RAINY = 0, 1, 2
LABELS = ["Sunny", "Cloudy", "Rainy"]

# Transition matrix: T[i][j] = P(next = j | current = i).
T = [
    [0.7, 0.2, 0.1],    # from Sunny
    [0.2, 0.7, 0.1],    # from Cloudy
    [0.1, 0.2, 0.7],    # from Rainy
]

def step_once(dist):
    # One step: dist' = dist @ T (a 1x3 vector times a 3x3
        matrix).
    return [
        sum(dist[i] * T[i][j] for i in range(3))
        for j in range(3)
    ]

# Start from a delta on Sunny and run 50 steps.
dist = [0.0, 0.0, 0.0]
dist[SUNNY] = 1.0
for _ in range(50):
    dist = step_once(dist)

for label, p in zip(LABELS, dist):
    print(f"{label}: {p:.4f}")
```

Listing 6: The same weather Markov chain in plain Python.

These programs share a common pattern. The parts of a randomised algorithm that are tedious in a general-purpose language (the round count, the union bound, the matrix multiplication, and floating-point comparisons) have been pushed into the language, and the parts that belong to the algorithm itself (the witness check, the vote, the transition) are what remain in the source.

3 Implementation

This chapter discusses the parts of building YAPPL that were technically interesting: the parser combinator architecture and the trickier productions it had to express, the runtime value system that grew alongside the language, the visualiser and its dual rendering backends, the testing strategy and its limitations, and the deployment of the interpreter as a sandboxed web playground. The goal is not to walk through every file in the source tree (that would duplicate the codebase itself) but to highlight the design decisions that were not obvious and the lessons that came out of making them.

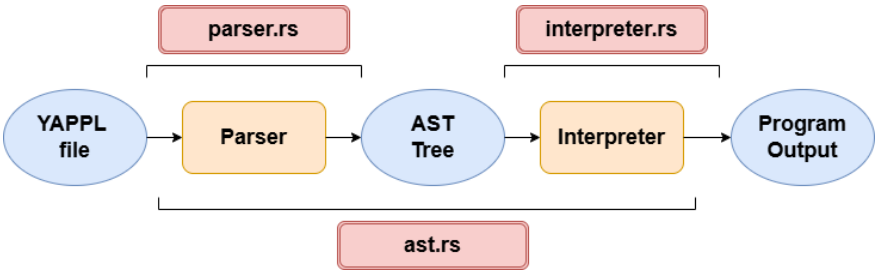

Figure 4: High-level structure of the YAPPL interpreter (`.rs` files are Rust source modules).

Figure 4 shows the overall pipeline: source text is read by the parser, converted into an AST, and then walked by the interpreter to produce an output stream that is rendered either to the terminal or to the web playground.

3.1 Parsing, interpreting, and visualising

The remainder of this section walks through the interesting implementation details of the three core modules that make up the pipeline of Figure 4: the parser that turns source text into an AST, the tree-walking interpreter that evaluates it, and the visualiser that renders distribution values into either ASCII histograms or inline SVG.

3.1.1 The parser

YAPPL's parser is built on the `nom` parser combinator library [10], which represents a parser as a Rust function of type `&str -> IResult<&str, T>` and provides combinators (`alt`, `many0`, `delimited`, `terminated`, ...) for composing them. The benefit of this style is that the parser reads almost as a transliteration of the BNF; the cost is that it

is essentially recursive descent, and so direct left recursion has to be rewritten and ambiguous productions have to be ordered carefully [16].

Two productions caused most of the parser-side difficulty. The first was the chain of postfix method calls (`die.sample().mean()`), which has to be parsed after any primary expression: a literal, a parenthesised expression, a constructor call, or a variable. The cleanest implementation turned out to be to factor the grammar into a `primary_term` that knows nothing about methods and a `primary_postfix` that loops over zero or more `(. | :) ident ( arg_list )` suffixes; this allowed both the new dot syntax and the legacy colon syntax (which appeared in earlier sample programs) to be supported uniformly without code duplication.

The second was the approximate equality operator. `~=` has to be tried before `==`, otherwise the leading `~` would be consumed as an unknown character and the whole comparison would fail with an unhelpful error. Once `~=` matches, the parser also has to look ahead optimistically for an optional `within <expr>` clause that supplies a tolerance. Bundling all of this into one AST node (`Expr::ApproxEq(Box<Expr>, Box<Expr>, Option<Box<Expr>>)`) kept the interpreter side simple at the cost of a slightly bespoke parsing rule.

A more pervasive complication was the special-form treatment of `bind`, `step`, `map`, and `distribution_of`. Each of these takes a bare function name as one of its arguments rather than a value, because YAPPL deliberately does not have first-class functions: function names live in a separate environment from variable bindings. This is the appropriate choice for keeping the runtime simple, but it means these constructs cannot be parsed as ordinary function calls. Each one has its own dedicated production in the grammar and its own evaluation path in the interpreter, where the function name is read as an identifier and looked up directly in the function table rather than evaluated as an expression.

Top-down recursive descent with backtracking was kept throughout; bottom-up parsers such as LR generators are more powerful in principle [2, 16], but for a small language with shallow expression trees the cost of backtracking is negligible and the iteration time savings from being able to read the parser as the BNF were substantial. This trade-off was first observed in the progress report and the year that followed gave no reason to revisit it.

```rust
/// After a primary expression, optionally consume chained `.method(args)` or `:method(args)` calls.
fn parse_primary_with_postfix(input: &str) -> IResult<&str, Expr> {
    let (input, base) = parse_primary_term(input)?;
    let (input, methods) = many0(alt((parse_dot_method, parse_colon_method)))(input)?;
    let mut result = base;
    for (method, args) in methods {
        result = Expr::ExprMethodCall { expr: Box::new(result), method, args };
    }
    Ok((input, result))
}

fn parse_dot_method(input: &str) -> IResult<&str, (String, Vec<Expr>)> {
    let (input, _) = eat_ws(tag("."))(input)?;
    let (input, method_name) = eat_ws(parse_identifier)(input)?;
    let (input, args) =
        delimited(eat_ws(tag("(")), parse_arg_list_optional, eat_ws(tag(")")))(input)?;
    Ok((input, (method_name.to_string(), args)))
}
```

Figure 5: The factoring of postfix method calls in `src/parser.rs`. `parse_primary_with_postfix` first delegates to `parse_primary_term` for the base expression (which knows nothing about methods), then uses the `many0(...)` combinator to consume zero or more method suffixes; inside the loop, `alt((parse_dot_method, parse_colon_method))` tries the dot syntax first and falls back to the legacy colon syntax. Each suffix parser uses `tag(".")` or `tag(":")` for the punctuation, `eat_ws(parse_identifier)` for the method name, and `delimited(eat_ws(tag("(")), parse_arg_list_optional, eat_ws(tag(")")))` for the argument list, so both syntaxes share the same factoring without code duplication.

3.1.2 The interpreter

The interpreter is a tree-walker that performs a depth-first traversal of the AST (Figure 6), evaluating leaves first and propagating values up. The most consequential design change between the progress report and the final language was the move from a bare `f64` runtime value to a tagged `RuntimeValue` enum with seven variants: `Int`, `Float`, `Bool`, `Dist`, `Certain`, `Uncertain`, and `Info`. The `Certain` and `Uncertain` wrappers are only ever produced as the return value of a single round of a probabilistic function, and `Info` is only produced by the pb-call assignment that bundles the round count and final confidence alongside the result. Encoding these as ordinary `RuntimeValue` variants meant that the rest of the interpreter (output, environment lookup, method dispatch) could remain agnostic about the new probabilistic machinery, which was important for keeping the diff manageable as the language grew.

The pb-call statement is at the centre of the runtime. When the interpreter sees `let r, info = f(args) with confidence >= c`, it reads the declared error class of `f`, computes the round count k from the appropriate formula ($\lceil -\log_2(1-c) \rceil$ for RP/coRP [27], $\lceil -8\ln(1-c) \rceil$ for BPP [20, 19]), and runs the function body k times in a loop, collecting the per-round returns. For RP and coRP a

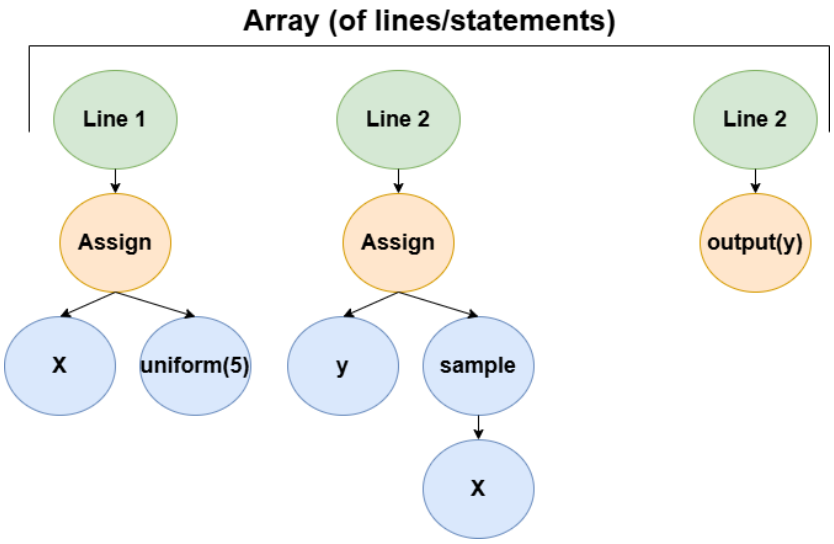

Figure 6: Shape of the AST consumed by the interpreter. The top-level statement list is an array, while expressions form trees evaluated bottom-up by depth-first traversal.

single `Certain` short-circuits the loop with confidence 1.0; for BPP all k rounds are run and the answer is the majority vote. The arithmetic lives in exactly one place, which means that fixing a bug in the round count fixes it for every program in the language at once.

A second consequence of this design is that forward references work without any additional bookkeeping. The top-level runner does two passes over the program items: a registration pass that collects every `fn` and `pb function` into the environment, and an execution pass that runs each statement in order. This means that a pb function can call another function defined further down the file, and a `distribution_of` call can name a function that has not yet been encountered in source order. The two-pass structure costs almost nothing for a language of this size but removes a class of confusing errors.

The `bind` and `step` primitives produce a `DynDist`, which is a dynamic discrete distribution over arbitrary runtime values. This is distinct from the AST-level `Dist::Discrete` because its keys can be enum variants rather than expressions. Merging outcomes during `bind` uses a `type::variant` key so that two paths that arrive at the same enum state collapse into a single bar in the resulting distribution; without this, the histograms produced by `step(..., n)` would explode in size with n instead of staying bounded by the

size of the state space.

3.1.3 The visualiser

Visualisation is implemented in its own module (`src/visualiser.rs`) and is structured around a single `HistogramData` struct and two render functions: `render_cli` for ASCII bars and `render_svg` for inline SVG. The interpreter's output buffer was widened from `Vec<String>` to `Vec<OutputLine>` so that text and histograms could be interleaved in a single output stream and rendered differently depending on the context. The CLI binary renders both as text; the web playground renders text into `<pre>` blocks and histograms into inline `<svg>`. The `run_to_string` entry point used by the unit tests goes through the CLI renderer so that test expected files are simple ASCII.

The most subtle issue here was that inline SVG elements share an HTML document namespace, so the gradient definitions used by `render_svg` have to carry unique identifiers when multiple histograms appear on the same page. The renderer takes an `idx` parameter that is threaded through from the playground's outer loop and used as a suffix on every gradient `id`. This is invisible to the user but caused several rounds of confused debugging when the second histogram on a page mysteriously inherited the first one's colour ramp.

For discrete distributions the visualiser computes the full probability mass function analytically using the same `get_dist_outcomes` routine that powers `expect`, so the picture is exact rather than sampled. For continuous distributions, where the PMF is not well-defined, the visualiser falls back to displaying the key parameters (min, max, and mean) with a gradient range bar. This is admittedly a placeholder for a true PDF plot, and is one of the extensions discussed in Chapter 4.

3.2 Testing and Evaluation

YAPPL has a unit test suite in `src/tests.rs` which is driven by sidecar `.expected` files: each program in `Sample/Deterministic/Passing/` is paired with a file containing its exact expected stdout, and each program in `Sample/Deterministic/Failing/` is paired with a file containing a substring that the resulting error message must contain. A pair of macros (`passing_test!` and `failing_test!`) generates a `#[test]` function for each program, so adding a new test case is a one-line change. At the time of writing the suite covers

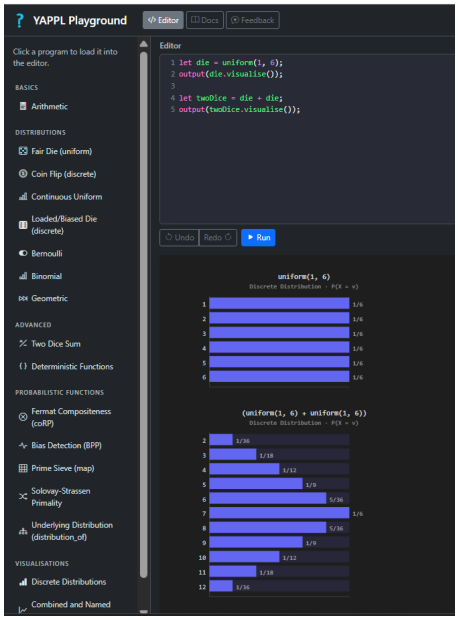

Figure 7: *
(a) Web playground (inline SVG)

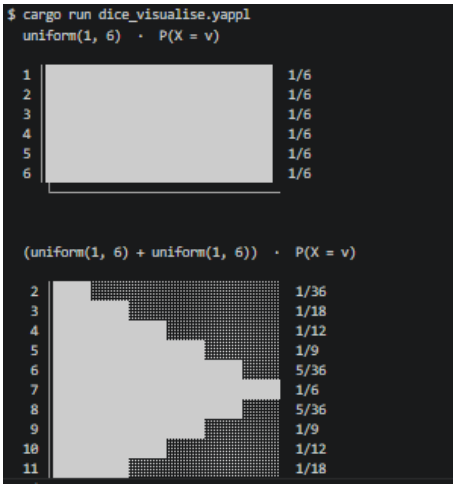

Figure 8: *
(b) CLI binary (ASCII)

Figure 9: The same two-line program (`uniform(1,6).visualise()` followed by `(die + die).visualise()`) rendered by both backends of the visualiser. The web playground (a) emits inline SVG with distinct colour gradients per histogram, while the CLI binary (b) emits the same probability mass functions as ASCII bars in the terminal. Both renderings are computed analytically from the same `HistogramData` struct and share the same probabilities; only the rendering backend differs.

24 passing programs and 10 failing programs, exercising arithmetic, comments, boolean expressions, control flow, regular and probabilistic functions, distribution constructors, distribution arithmetic, distribution equality, `map`, and full Markov chain evolution. The failing-test set is just as important as the passing one: it pins down the exact error messages produced by the parser and interpreter for things like missing semicolons, undefined variables, type mismatches, wrong argument counts, division by zero, and pb function calls that omit the `with confidence` clause.

Many of the test programs and expected files were drafted by an LLM and then audited by hand. Hand-writing dozens of small programs and computing their expected output is mechanical work, and an LLM is well-suited to generating both halves from a one-line description. Every generated expected file was checked against the actual interpreter output before being committed, and a handful of programs that exercised nothing useful were discarded; the LLM

functioned as a typing aid rather than as a source of correctness.

The conspicuous absence in the test suite is the probabilistic sample programs in `Sample/Probabilistic/`. These cannot be compared against an exact expected file because the runtime samples real random witnesses, so the round count and the printed `Info` struct are not deterministic. A natural extension would be to seed the RNG from a fixed value and assert against the resulting deterministic trace, or to assert statistical properties (for example, that running `is_prime(53)` 1000 times produces `true` every time, and that `is_prime(91)` produces `false` more than 95% of the time). Either approach would be straightforward to add; neither was a priority for this project relative to broadening the language itself.

Beyond unit testing, the project was made available to human users via a feedback form linked from the playground. The intent was acceptance testing: could a non-author write a YAPPL program, and what did they think of the experience? The responses were of limited use, since visitors were not specialists in probabilistic programming, so the questions they answered well (such as whether the website was visually appealing) were not the ones the project wanted answered (such as whether the pb function abstraction matched how they would write a randomised algorithm by hand). The conclusion is that meaningful acceptance testing of a small language like YAPPL needs either expert respondents or carefully designed task-based studies; an open-ended feedback form linked from a public URL produces only the responses of non-experts, and those responses say little about language design.

A more useful qualitative measure of success was the project's own original objectives. The specification asked for a language specification, an interpreter, and an evaluation of language features. The first is provided as an appendix, the second is the codebase, and the third is Chapter 2. Against the more concrete demands of the introduction (mechanisms for analysing confidence, distributions as first-class values, and compositional combinators) each is now realised by a specific feature (`pb function` and `distribution_of`, the distribution constructors and method set, and `bind`, `step`, and `map` respectively). The most visible gap is that BPP-style two-sided amplification is hard-coded for per-round success $p = 3/4$; a real BPP algorithm with a different bias needs a different Chernoff constant [20], and parametrising this is one of the items in Chapter 4.

3.3 Interface and Deployment

YAPPL ships as both a CLI binary (`yappl <file>`) and a web playground. The live playground is hosted at `https://yappl.containers.uwcs.co.uk/`, and the full source for the interpreter, the playground, and the deployment configuration is on GitHub at `https://github.com/RexMortem/3YP`; the repository's README contains step-by-step instructions for building the interpreter locally, running the test suite, and reproducing the deployment from the published container image. The remainder of this subsection describes the playground front end and the API it talks to, the Docker packaging and CI/CD pipeline that produces the deployed image, and the host on which the image runs.

3.3.1 The playground front end

The playground is the default user interface: a single page with a code editor on the left, an output pane on the right, and a sidebar of preloaded sample programs that also serve as an introduction to the language's features. Submitting a program POSTs the source text to a `/run` endpoint on the server; the server runs the program through the interpreter inside the same process and returns either an HTML response (containing `<pre>` blocks for text output and inline SVG for histograms) or a plain-text error. The client uses `innerHTML` for HTML responses and `textContent` for errors, which means the SVG histograms appear directly in the page without any extra plotting library. There is no I/O in the language and no persistent state, so the server is fully stateless and a request can be handled without any sandboxing beyond the natural limits of running another process: there are no file reads, no network calls, and no way for a program to escape the interpreter.

A small number of accessibility features have been added to the playground for visitors who are not the primary target audience of the site. The page supports both light and dark modes, with a toggle in the navbar that switches both the page chrome and the editor's syntax-highlighting theme (CodeMirror's `eclipse` theme in light mode and `dracula` in dark mode); the chosen mode is remembered between visits via `localStorage`. The editor toolbar also exposes explicit *undo* and *redo* buttons next to the *Run* button. These are functionally redundant on a desktop, where `Ctrl+Z` and `Ctrl+Shift+Z` already work, but they were added for mobile browsers since you can't perform those shortcuts on a phone and a programmer who is exploring the language on a phone or tablet

would otherwise have no way to undo a mistaken edit.

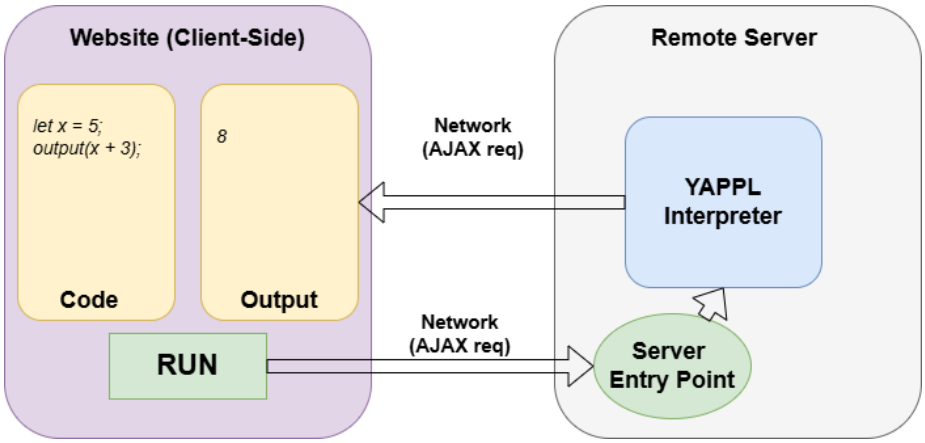

Figure 10: Architecture of the YAPPL platform: the browser editor sends source text to the `/run` endpoint on the server, which invokes the in-process interpreter and returns rendered HTML (with inline SVG histograms) for display in the output pane.

Figure 10 shows how the pieces fit together. This diagram was originally drafted in the progress report as a sketch of a hypothetical platform; it now describes the deployed system unchanged.

3.3.2 Packaging and CI/CD

The whole stack is packaged as a Docker image. The `Dockerfile` performs a multi-stage build (a Rust builder stage that compiles the interpreter and the web binary, and a slim runtime stage that contains just the binary and the static assets) so the resulting image is small enough to deploy comfortably. There are two GitHub Actions workflows backing this. The first, `docker-publish.yml`, runs on pushes to `main`, builds the image, and pushes it to GitHub's container registry (`ghcr.io`) tagged both with the commit SHA and with `latest`. The second, `test.yml`, runs on every push to any branch and executes `cargo test`; this CI/CD step catches regressions before they reach `main`, and was added once the test suite grew large enough that running it locally before every push had become inconvenient. Caching the cargo registry between runs (via the `Swatinem/rust-cache` action) brings a clean test run to roughly a minute.

Figure 11 shows what this looks like in practice: when a commit fails the unit tests on `main`, the failure is surfaced directly on the commit page in GitHub, which makes it impossible to miss and

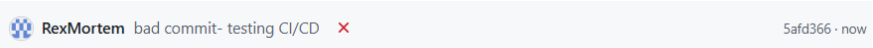

Figure 11: A failing commit on the `main` branch as it appears on GitHub. The `test.yml` workflow has run `cargo test` against the new commit, the unit tests have failed, and the failure is reported directly against the commit; this prevents broken code from being silently deployed.

gives an immediate signal that the build needs fixing before the next deployment.

3.3.3 Hosting

The image is deployed onto the University of Warwick Computing Society's Portainer service; the README on the GitHub repository contains the exact stack file and the steps to update it from the latest `ghcr.io` tag. Portainer was chosen because it is free to UWCS members, was already running on infrastructure that the author had access to, and needed nothing more than a container image and a port to host the playground. The deployment is therefore reproducible: anyone with access to the same Portainer (or any other Docker host) can run the latest tagged image and obtain an identical playground. This was a deliberate goal: a project whose deployment is a single tagged image is one that a future reader can actually run.

4 Future Work

YAPPL in its current state is a usable prototype for writing and analysing randomised algorithms, but the design choices made along the way (and the limited time available within a third-year project) have left several promising directions unexplored. This chapter discusses how the existing work could be built on rather than what was simply left undone, with a focus on extensions that would meaningfully push the language forwards.

4.1 A Bayesian Extension to Confidence Amplification

The most interesting limitation of the current confidence model is that it is purely frequentist (it reasons only about long-run rates of correctness over many independent runs, with no factoring in of the programmer's own belief about the true underlying error rate) and assumes the worst case. The runner for an RP function such as Solovay-Strassen treats the per-round error probability as exactly $1/2$, because this is the bound proved by the algorithm [27]. After k rounds with no `Certain` answer, the reported confidence is therefore $1 - (1/2)^k$, which is a true lower bound but is often very loose: for almost all composite numbers the actual fraction of fooling witnesses is far smaller than $1/2$, and is often closer to $1/4$ or $1/8$ [20, 23].

A Bayesian extension would let YAPPL reason about its posterior belief given what it has actually observed. The Bayesian vocabulary distinguishes a prior belief (what was held before seeing any evidence) from a posterior belief (what is held after taking the evidence into account). For example, if the prior probability that a given algorithm has a low per-round error rate is modest, then observing many rounds of correct behaviour shifts the posterior to a much stronger belief in a low error rate than the prior alone would justify. Rather than fixing the per-round error rate at the worst case, the runner would maintain a distribution over the unknown error rate p and update that distribution from prior to posterior after each round. The natural choice for this prior is the **Beta distribution**. A Beta(α, β) prior over a Bernoulli success probability is a conjugate prior [14], meaning that after observing successes and failures the posterior is itself a Beta distribution, with parameters incremented by the observed counts. Conjugate priors are valuable because the update is in closed form: there is no need to run any kind of approximate inference to keep track of the belief, which is exactly the property a confidence-amplification loop wants. YAPPL already implements the Beta distribution as a runtime type for the

`distribution_of(..., bayesian, N)` feature, so much of the supporting machinery is in place.

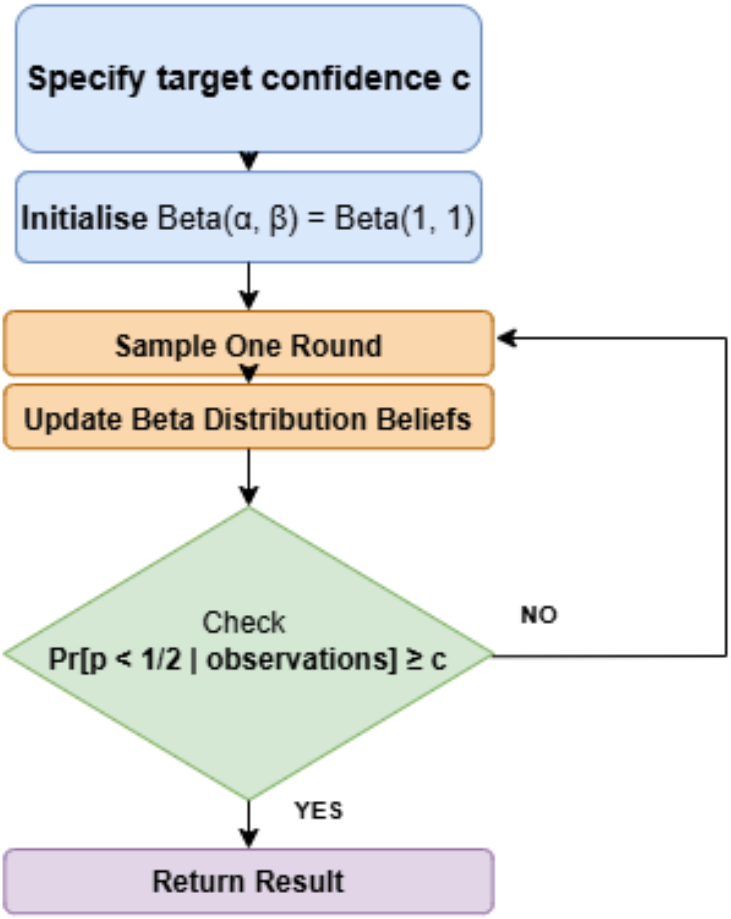

Figure 12: The Bayesian update loop. The programmer specifies a target confidence c; the runtime initialises a flat $\mathrm{Beta}(1, 1)$ prior over the unknown per-round error rate p, then repeatedly samples one round of the probabilistic function, updates the Beta posterior by incrementing α on a `Certain` return and β on an `Uncertain` one, and checks whether $\Pr[p < 1/2 \mid \text{observations}] \geq c$ under the current posterior. The loop terminates as soon as that threshold is reached and the result is returned alongside the round count.

The benefit is that, in the common case, fewer rounds are needed to reach a given confidence target: if the algorithm has so far been correct on every round, the posterior will quickly concentrate well below $1/2$ and the loop will terminate early. The cost is that the bound is no longer worst-case and so depends on the prior: a careless choice of prior (or an adversarial input) could overstate the runner's confidence, and the implementation would need to carefully docu-

ment its assumptions. There is also a non-trivial implementation cost: numerical integration of the Beta posterior for the cumulative probability $\Pr[p < 1/2 \mid \text{observations}]$ is required, and this falls outside the closed-form arithmetic the rest of the interpreter currently uses. Within the time budget of a third-year project this was not feasible, but it would be a natural next step for a follow-on project.

4.2 Richer Continuous Distributions and Continuous-Time Markov Chains

YAPPL currently supports a small set of continuous distributions (uniform, Beta) primarily for completeness rather than as a serious modelling tool. Adding a richer collection (Normal, Exponential, Gamma, Dirichlet) and the operations to combine them would open up several modelling applications that the language cannot currently express. Most notably, it would allow **continuous-time Markov chains**: chains where the dwell time in each state is itself a random variable drawn from an exponential distribution, rather than a fixed unit step [22]. The existing `bind` and `step` machinery for discrete-time Markov chains would generalise naturally, and the result would be expressive enough to model queueing systems, epidemic dynamics, and continuous-time decision processes. The principal challenge is that exact analytical computation is no longer possible for most combinations: the interpreter would need a hybrid mode that handles closed-form distributions analytically where it can and falls back to Monte Carlo sampling otherwise.

4.3 Correlated Rounds

A simplifying assumption pervades the entire confidence-amplification model: rounds of a probabilistic function are treated as completely independent. This is correct for the academic algorithms used as test cases, but real-world randomised systems often have correlation across rounds. Examples include sampling without replacement, reusing a shared random seed, and consecutive measurements that share some hidden state. A natural extension would let the programmer declare a correlation structure between rounds, and have the runtime adjust its confidence calculation accordingly. The most lightweight version of this would be to allow the programmer to attach a correlation coefficient ρ between consecutive rounds; a more ambitious version would allow an arbitrary covariance function. Implementing this would require generalising the round-counting formula, which currently assumes independence in its Chernoff [20]

and Geometric bounds, but the language-level surface could be very small: a single `with correlation` clause attached to a probabilistic function call.

4.4 Quantum Algorithms

Quantum algorithms were a tempting direction for this project but were ruled out as too large in scope. They would have required complex numbers as a primitive type, a new `Superposition` distribution that tracks amplitudes rather than probabilities, and an explicit `measure` operation that collapses a superposition to a classical outcome. There is a pleasing alignment between this and YAPPL's design philosophy: `measure` would naturally be the latest possible operation in a program, just as `sample` is in the existing language, which fits the principle of doing as much work as possible at the distribution level before committing to a concrete value. A future project could explore this (the abstract structure is very similar) but it would also require a substantial investment in understanding the underlying physics and the standard quantum algorithms (Deutsch-Jozsa [12], Grover, Shor) well enough to write meaningful test cases.

4.5 Implementation Improvements

Finally, the implementation itself has a number of avenues for improvement. The parser is a hand-written recursive descent parser using `nom` [10], which is well-suited to the size of the language but is theoretically less powerful than a bottom-up LALR or GLR parser [2, 16]; if the language grew significantly more complex, switching parser style would become attractive. The interpreter is a tree-walking evaluator, which is simple and easy to extend but is the slowest standard interpreter style. Most modern interpreters use some form of compilation: bytecode generation followed by a virtual-machine dispatch loop, or a tracing JIT, both of which obtain order-of-magnitude speedups while keeping the implementation tractable [2, 4, 13]. A bytecode back-end would be a natural next step if YAPPL were ever pushed to handle larger workloads.

5 Conclusion and Reflection

This project set out to design and implement a probabilistic programming language with first-class support for the construction and analysis of randomised algorithms, and to evaluate which language features actually make probabilistic programming more ergonomic. The original specification [11] (Appendix B) asked for three things: a specification of the language, an interpreter, and an evaluation of language features, all three of which have been delivered. The grammar and semantics are documented in the appendix, the interpreter is a tagged tree-walker written in Rust, deployed as a public web playground packaged into a single Docker image, and the language features are evaluated against concrete sample programs in Chapter 2. These include coin-flip RP and coRP examples, the BPP bias detector, the Markov-chain weather model [22], and the distribution algebra inspired by Haskell's `Data.Distribution` [1].

Against the more concrete demands of the introduction, each is realised by a specific feature. "Mechanisms for analysing probabilistic algorithms" is realised by `pb function` and `distribution_of`. "Modelling underlying distributions" is realised by the distribution constructors and method set. "Combining distributions and probabilistic algorithms in a meaningful way" is realised by `bind`, `step`, and `map`. The most visible gap is that BPP-style amplification is hard-coded for per-round success $p = 3/4$ (the standard assumption [20, 26]) and that the confidence model is purely worst-case; both gaps are directly addressed in Chapter 4.

Table 1 summarises the alignment between the original specification and the delivered language at a glance.

On a personal note, the project taught me more about programming language design and managing a long, difficult project than I expected at the outset. The hard problem turned out not to be building a parser or an interpreter but deciding which ideas to include or remove, and specifically how the language should work. The most useful lesson was that pruning features improves a small language more quickly than adding them: every feature considered and discarded (Prolog-style unification, built-in confidence values on every variable, quantum superpositions) clarified the purpose of the features that remained.

Original objective (from [11])	**Concrete YAPPL feature(s) that satisfy it**
Produce a specification for the language	The language specification (Appendix A), including a BNF grammar and small-step operational semantics.
Build an interpreter for the language	A tagged tree-walking interpreter written in Rust (Chapter 3), with a CLI binary and a public web playground packaged as a single Docker image.
Evaluate language features for probabilistic programming	The "Was this ergonomic?" discussions in Chapter 2, supported by the sample programs under `Sample/Probabilistic/` and the unit test suite.
Mechanisms for analysing probabilistic algorithms	`pb function` (with declared error class), confidence-target call sites, and `distribution_of` in its analytical, empirical, and Bayesian modes.
Modelling underlying probability distributions as first-class values	Distribution constructors (`uniform`, `uniformContinuous`, `Bernoulli`, `Binomial`, `Geometric`, `Discrete`, `Beta`) and the method set `sample`, `expect`, `mean`, `min`, `max`, `visualise`.
Combining distributions and probabilistic algorithms compositionally	Lifted `+` for analytical convolution, `bind` and `step` for typed Markov chains over user-defined enums, and `map` with a shared confidence budget split via the union (Bonferroni) bound.

Table 1: Alignment between the original project objectives and the concrete YAPPL features that satisfy them. The top three rows are the explicit objectives from the project specification; the bottom three are the more concrete demands stated in the introduction (Chapter 1).

References

[1] Haskell data.distribution. `https://hackage.haskell.org/package/distribution-1.1.1.0/docs/Data-Distribution.html`. Accessed: 2026-01-20.

[2] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Pearson, 2 edition, 2006.

[3] Lennart Augustsson, Joachim Breitner, Koen Claessen, Ranjit Jhala, Simon Peyton Jones, Tiark Rompf, Ilya Sergey, Arnaud Spiwack, and Sam Tobin-Hochstadt. The Verse calculus: A core calculus for functional logic programming. *Proceedings of the ACM on Programming Languages*, 7(ICFP):417–447, 2023.

[4] John Aycock. A brief history of just-in-time. volume 35, pages 97–113, 2003.

[5] Eli Bingham, Jonathan P. Chen, Martin Jankowiak, Fritz Obermeyer, Neeraj Pradhan, Theofanis Karaletsos, Rohit Singh, Paul Szerlip, Paul Horsfall, and Noah D. Goodman. Pyro: Deep universal probabilistic programming. *Journal of Machine Learning Research*, 20(28):1–6, 2019.

[6] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[7] Carlo E. Bonferroni. *Teoria statistica delle classi e calcolo delle probabilità*. Pubblicazioni del R Istituto Superiore di Scienze Economiche e Commerciali di Firenze, 1936.

[8] William F. Clocksin and Christopher S. Mellish. *Programming in Prolog*. Springer, 5 edition, 2003.

[9] Graham Cormode and S. Muthukrishnan. An improved data stream summary: The count-min sketch and its applications. *Journal of Algorithms*, 55(1):58–75, 2005.

[10] Geoffroy Couprie and nom contributors. nom: A byte-oriented, zero-copy, parser combinators library. `https://github.com/rust-bakery/nom`, 2024. Accessed: 2026-04-09.

[11] Edward Denton. CS351 project specification: A probabilistic programming language. University of Warwick, Department of Computer Science, 2025. Internal project specification.

[12] David Deutsch and Richard Jozsa. Rapid solution of problems by quantum computation. *Proceedings of the Royal Society A*, 439(1907):553–558, 1992.

[13] L. Peter Deutsch and Allan M. Schiffman. Efficient implementation of the Smalltalk-80 system. In *Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*, pages 297–302, 1984.

[14] Andrew Gelman, John B. Carlin, Hal S. Stern, David B. Dunson, Aki Vehtari, and Donald B. Rubin. *Bayesian Data Analysis*. Chapman and Hall/CRC, 3 edition, 2013.

[15] David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 23(1):5–48, 1991.

[16] Dick Grune and Ceriel J. H. Jacobs. *Parsing Techniques: A Practical Guide*. Springer, 2008.

[17] Nikolay Ivanov. Is rust c++-fast? benchmarking system languages on everyday routines, 09 2022.

[18] Avani Jindal, Janhvi Joshi, Nikhil Sajwan, Naman Adlakha, and Sandeep Pratap Singh. Efficient use of randomisation algorithms for probability prediction in baccarat using: Monte carlo and las vegas method. In Mayank Dave, Ritu Garg, Mohit Dua, and Jemal Hussien, editors, *Proceedings of the International Conference on Paradigms of Computing, Communication and Data Sciences*, pages 91–103, Singapore, 2021. Springer Singapore.

[19] Michael Mitzenmacher and Eli Upfal. *Probability and computing: randomization and probabilistic techniques in algorithms and data analysis*. Cambridge University Press, Cambridge, second edition, 2017.

[20] Rajeev Motwani and Prabhakar Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.

[21] Aditya V. Nori, Chung-Kil Hur, Sriram K. Rajamani, and Selva Samuel. R2: An efficient MCMC sampler for probabilistic programs. In *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence*, pages 2476–2482. AAAI Press, 2014.

[22] James R. Norris. *Markov Chains*. Cambridge University Press, 1997.

[23] Michael O. Rabin. Probabilistic algorithm for testing primality. *Journal of Number Theory*, 12(1):128–138, 1980.

[24] Jacob T. Schwartz. Fast probabilistic algorithms for verification of polynomial identities. *Journal of the ACM*, 27(4):701–717, 1980.

[25] N. Siddharth, Brooks Paige, Jan-Willem van de Meent, Alban Desmaison, Noah D. Goodman, Pushmeet Kohli, Frank Wood, and Philip H. S. Torr. Learning disentangled representations with semi-supervised deep generative models. In *Advances in Neural Information Processing Systems 30 (NeurIPS 2017)*, pages 5925–5935, 2017.

[26] Michael Sipser. *Introduction to the Theory of Computation*. Cengage Learning, 3 edition, 2012.

[27] R. Solovay and V. Strassen. A fast Monte-Carlo test for primality. *SIAM Journal on Computing*, 6(1):84–85, 1977.

A YAPPL Language Specification

The full standalone YAPPL language specification is included on the following pages. It defines the lexical structure, the deterministic core, the probabilistic features (`pb function`, confidence amplification, `distribution_of`, the Markov chain primitives), the type system, the BNF grammar, a small-step operational semantics, and a reference catalogue of the standard library.

YAPPL

A Language Specification for
A Probabilistic Programming Language

Contents

1 Introduction and Design Pillars

Why a probabilistic programming language?

It is tempting to think that randomness is something a careful programmer should try to avoid. Unpredictable behaviour is generally undesirable in software, and a great deal of programming-language research has been directed at making programs easier to reason about by reducing the amount of non-determinism they exhibit.

Despite this, randomness is one of the most useful tools in modern algorithm design. By allowing an algorithm to make random choices, we can avoid worst-case behaviour, simplify the design of the algorithm, and achieve performance that is not available to any purely deterministic approach. Many of the most heavily used algorithms in practice (sorting, hashing, primality testing) depend on probability for guarantees that hold with high probability rather than absolute certainty. There are entire complexity classes, most famously **RP**, of problems solvable efficiently by a probabilistic Turing machine but not known to be solvable efficiently by a deterministic one. The trade-off that probability introduces between correctness and performance is part of what makes the algorithms work, and it can usually be tuned by running more iterations at the cost of more time.

Given how central this trade-off is, it would be useful to have a programming language whose semantics reflect the trade-off directly rather than leaving it to the programmer to manage by hand. In particular, such a language should provide:

- mechanisms for analysing probabilistic algorithms,
- the ability to name and manipulate the underlying probability distributions, and
- the ability to combine distributions and probabilistic functions in a meaningful way.

YAPPL (*Yet Another Probabilistic Programming Language*) is that language. It is a statically-typed language aimed primarily at Monte Carlo algorithms: computations where the *shape* of the error is known in advance, and the question is how aggressively to amplify it to reach a confidence level the programmer is comfortable with. A Solovay–Strassen primality tester, a bounded-error majorityvote, a Markov-chain step, and a biased-coin flip are all expressible in YAPPL, and all of them reuse the same small set of primitives.

The defining feature of YAPPL is that *probability distributions are values*. Writing

```
let die = uniform(1, 6);
let p = die.expect(6);
let roll = die.sample();
```

is the same kind of statement as binding an integer. The distribution `die` can be passed around, combined with another distribution using `+`, queried analytically with `expect`, sampled with `sample`, or visualised. The dissertation accompanying this specification summarises this idea as *"obtain values as late as possible"*: most useful work can be done at the level of the distribution itself, and turning a distribution into a concrete sample is something to defer until the programmer needs a particular answer. Much of the rest of the language follows from taking that idea seriously.

1.1 Design Pillars

The language rests on five design pillars. Each is stated here in plain terms, with forward references to the section that develops it formally. Readers familiar with functional-programming languages will recognise many of these ideas as specialisations of patterns that already work well for deterministic computation, including function composition, monadic bind, and structural equality. A guiding intuition behind YAPPL is that the same patterns, applied to distributions, give a language in which probabilistic reasoning is as natural as arithmetic.

(a) Distributions are values, and we work on them for as long as we can. A `Dist<T>` is indistinguishable, syntactically, from any other value in the language: it can be bound, passed, returned, compared, and stored. Crucially, *distributions carry their full algebraic structure*. Adding two distributions with `+` is convolution, computed analytically; the `bind` operation (§4) is the monadic bind for distributions, mirroring function composition in a functional language; exact equality is structural, and approximate equality (using total-variation distance or moment comparison) is built-in with a programmer-controllable tolerance. The only operation in the language that crosses from the algebraic world to the stochastic one is `sample`, and it is used as a last step rather than a first one. This is the "obtain values as late as possible" slogan made concrete.

An earlier design considered baking a confidence value into *every* non-distribution value, so that doubling a value with confidence x would produce one with confidence x^2. This was rejected as "too cumbersome and likely to cause an unnecessarily complicated language". Making distributions themselves the unit of reasoning is the cleaner alternative: there is no need to track a confidence alongside every integer, because if a quantity has interesting probabilistic behaviour it should be a distribution in the first place.

(b) Probabilistic functions declare their error in their signature. A regular function has a signature of the form `fn f(x: T)-> U`. A probabilistic function extends this with an *error class* and an *error distribution* that together describe what its per-round behaviour looks like:

```
pb function is_prime(n: int) -> bool {
    error_class: RP,
    error_distribution: Geometric
} {
    ...
}
```

The error class is one of `RP`, `coRP`, or `BPP`, following the standard randomised-complexity taxonomy: `RP` means "a `false` answer is always correct; a `true` answer may be wrong", `coRP` is its mirror, and `BPP` means "both answers may be wrong, but with bounded bias". The error distribution describes *how* the per-round error compounds under repetition: typically `Geometric` for the one-sided classes (where each round halves the residual error), and `Binomial` for BPP (where the correct answer is recovered by a majority vote across rounds).

These annotations are not documentation: they are part of the function's type (§5) and drive the compiler's choice of repetition strategy. Two functions that differ only in their error class are different types.

(c) The compiler derives the repetition strategy from the error class. Once the error class is known, the correct way to amplify a single round into a confident answer is no longer a programmer choice: it is a fact about the class. YAPPL exploits this by moving the repetition logic out of user code entirely:

- **RP**: run rounds in sequence, short-circuit as soon as a `Certain` answer appears, otherwise accumulate `Uncertain` rounds until the combined error bound $(1/2)^k$ is small enough.
- **coRP**: the dual of RP, with "certain" and "uncertain" swapped and the same geometric decay.
- **BPP**: run a fixed number of rounds and return the majority; the required number of rounds is the smallest k for which the Chernoff bound $e^{-k/8}$ meets the confidence target.

The programmer writes the single-round body once, declares the error class, and stops there. The amplification loop lives in the compiler. This avoids one of the traditional awkwardnesses of writing Monte Carlo code, where every user of a probabilistic function ends up re-implementing the same "run it k times and do something sensible" wrapper.[41]

(d) Confidence targets are the user-facing abstraction. The programmer never says "run 37 rounds". The programmer says *how confident they want to be*:

```
let result, info = is_prime(53) with confidence >= 0.99;
```

The language translates the target into the smallest number of rounds sufficient under the repetition strategy chosen in pillar (c), runs them, and binds the answer together with an `info` record reporting the actual number of rounds used and the actual (possibly higher) confidence achieved.

The significance of this inversion (targets in, iterations out) is that it decouples authors of probabilistic functions from their callers. The author of `is_prime` does not have to anticipate how confident its users will want to be, and users who need a higher confidence simply raise the target. For multi-call blocks where several probabilistic answers must all be simultaneously correct, the same mechanism distributes the error budget across the calls via the union (Bonferroni) bound, discussed in §4, so the programmer never performs the union bound by hand.

(e) Distributions and probabilistic functions are dual views of the same object, as much as possible. For every probabilistic function `f` there is an implicit distribution describing what a single round of `f` does: how often a certain answer is produced, how often an uncertain one, and (in the Bayesian mode) how confident we should be in that ratio after finitely many observations. YAPPL provides features that attempt to convert between the two views as far as is practical; the `distribution_of` operator reifies the implicit distribution of a probabilistic function into a first-class `Dist` value, and the `bind` and `step` operators run in the other direction by treating a distribution as the input to a transition function:

```
let d1 = distribution_of(is_prime(53), analytical);
let d2 = distribution_of(is_prime(53), empirical, 1000);
let d3 = distribution_of(is_prime(53), bayesian, 1000);
```

These are not three ways of asking the same question; they are three *different* questions that happen to share a syntactic form. The `analytical` form reports the distribution implied by the declared error class. The `empirical` form runs N single rounds and reports the observed certain-rate as a `Bernoulli(p)`. The `bayesian` form, given the same data, reports a Beta posterior over that rate. Each is a distinct view of the same underlying object, and all three produce ordinary `Dist<T>` values that can be combined, compared, and visualised like any other distribution.

The duality runs in the other direction too. A distribution can be viewed as a trivial probabilistic function that has a single round: sampling it once. The `bind` and `step` operators (§4) use this direction of the duality when iterating a Markov chain, by composing a transition function $S \to$ `Dist<S>` repeatedly with a distribution over states.

Taken together, pillars (a)–(e) describe a language in which theboundary between "an algorithm that happens to be probabilistic" and "a distribution that happens to be computable" is softened to the point of disappearing. Distributions give static structure; probabilistic functions give operational behaviour; and the `distribution_of`, `bind`, and `step` operators are the bridges between the two. The *obtain values as late as possible* slogan applies in both directions: if every intermediate step is a distribution, nothing is lost until a sample is finally drawn.

1.2 What this specification covers

The remainder of this document fixes YAPPL precisely enough to implement. Section 2 gives the lexical structure; Section 3 sketches the deterministic core, which is deliberately small so that the probabilistic features have room to breathe; Section 4 is the main content, covering the `Dist<T>` type, probabilistic functions, confidence targets and their Bonferroni distribution across multi-call blocks, `distribution_of`, the Bayesian extensions, and the Markov-chain primitives; Section 5 formalises the type system with error classes as part of function types; Section 6

walks through four worked examples; Section 7 is the full BNF grammar, kept in sync with the reference `grammar.bnf`; Section 8 gives a small-step operational semantics in which one rewrite step takes an expression to a distribution rather than to a single sampled value; and Section 9 is a reference catalogue of the standard library.

Typing rules throughout use the standard inference-rule notation with premises above the bar and the conclusion below. Inline code is in monospace; cross-references are clickable in the PDF.

2 Lexical Structure

This section fixes the tokens of YAPPL: what a YAPPL source file looks like to the tokeniser, before any grammar productions apply. The intent is that a compiler can produce a stream of tokens from raw text using only the rules in this section.

A YAPPL source file is a sequence of Unicode characters encoded in UTF-8. The tokeniser groups them into *tokens*, discarding whitespace and comments between tokens. Tokens fall into five categories: identifiers and keywords, literals, operators and punctuation, reserved symbols for probabilistic constructs, and end-of-statement markers.

2.1 Whitespace and Comments

Whitespace consists of the ASCII characters space ( ), tab (␣), carriage return, and line feed. Whitespace is *not* significant: it separates tokens but is otherwise discarded. Newlines do not terminate statements (only the semicolon does, see below).

YAPPL has one form of comment: a line comment introduced by `//` and extending to the next newline.

```
// This is a comment.
let x = 1; // Trailing comments are also fine.
```

Comments are treated as whitespace by the tokeniser.

2.2 Identifiers

An identifier begins with an ASCII letter or underscore and continues with any sequence of ASCII letters, digits, or underscores:

$$\textit{ident} \;::=\; \texttt{[a-zA-Z\_][a-zA-Z0-9\_]}^{*}$$

YAPPL distinguishes two lexical sub-classes of identifiers by their leading character. This distinction is enforced by the parser, not by the tokeniser, but it is visible in the grammar:

- **Lowercase-initial identifiers** (*lident*) are used for variables, function names, and the built-in scalar types `int`, `float`, `bool`.
- **Uppercase-initial identifiers** (*uident*) are used for enum type names and enum variants.

This discipline makes the grammar unambiguous without a symbol table: in a type position, `weather` is a syntax error but `Weather` is a valid named type (see §3). Reserved keywords (next subsection) are neither lidents nor uidents – they are their own tokens.

2.3 Keywords

The following identifiers are reserved and may not be used as variable or function names. They are split into three groups for readability, but the tokeniser treats them uniformly as fixed-keyword tokens.

Keywords are reserved in all positions; it is, for example, illegal to name a variable `map` or `bind` even when the context would not require those to be operators.

The distribution constructors `uniform`, `uniformContinuous`, `Discrete`, `Bernoulli`, `Binomial`, `Geometric`, and the built-in symbolic functions `jacobi` and `mod_exp` are *not* lexical keywords: they are ordinary identifiers that the parser resolves as special forms when they appear in call position. Shadowing them is allowed in principle but strongly discouraged.

Group	Keywords
Declarations	`fn`, `pb`, `function`, `enum`, `let`
Control flow	`if`, `else`, `return`, `output`
Scalar types	`int`, `float`, `bool`
Booleans	`true`, `false`
Operators	`mod`, `and`, `or`, `not`, `within`
Probabilistic core	`with`, `confidence`, `Certain`, `Uncertain`
Probabilistic ops	`map`, `distribution_of`, `bind`, `step`
`distribution_of` modes	`analytical`, `empirical`, `bayesian`
Error classes	`RP`, `coRP`, `BPP`
Error metadata fields	`error_class`, `error_distribution`

Table 1: Reserved keywords.

2.4 Literals

2.4.1 Integer and Float Literals

YAPPL distinguishes integer and float literals by the presence of a decimal point:

$$\begin{aligned} \textit{int\_lit} &::= \texttt{[0-9]}^{+} \\ \textit{float\_lit} &::= \texttt{[0-9]}^{+}\ \texttt{.}\ \texttt{[0-9]}^{+} \end{aligned}$$

An integer literal has type `int`; a float literal has type `float`. Neither form carries a sign: a leading minus is parsed as the unary negation operator, not as part of the literal. A trailing decimal point without fractional digits (e.g. `3.`) is a *lexical* error.

Integer literals are decoded as signed 64-bit integers; float literals are decoded as IEEE-754 double-precision values.

2.4.2 Probability and Confidence Literals

YAPPL does not have a distinct lexical form for probabilities, confidence targets, or tolerances: these are ordinary `float` literals and their well-formedness (e.g. $p \in [0, 1]$ for a Bernoulli parameter, or $c \in [0, 1)$ for a confidence target) is checked at the type level (§5) or at runtime, depending on the construct. Thus the following are all lexically indistinguishable:

```
let p = 0.3; // a probability
with confidence >= 0.99; // a confidence target
~= 0.42 within 0.01 // a tolerance
```

This uniform treatment keeps the lexer simple; the probabilistic meaning is carried by the *context* in which a float literal appears.

2.4.3 Boolean Literals

The two boolean literals are the keywords `true` and `false`.

2.5 Operators and Punctuation

YAPPL has the following operator tokens. They are listed here in decreasing precedence; the full precedence and associativity table appears in §7.

Punctuation tokens are: parentheses `( )`, curly braces `{ }`, square brackets `[ ]`, comma `,`, colon `:`, semicolon `;`, and the function-arrow digraph `->`. Angle brackets `< >` are *not* a separate pair of tokens: they are reused from the comparison operators, and their interpretation as type-argument delimiters (as in `Discrete<Weather>`) is a grammatical matter, not a lexical one.

Category	Tokens	Notes
Postfix (method call)	`.` `:`	`d.sample()`, `d:sample()`
Unary	`-` `!`	arithmetic negation, logical not
Multiplicative	`*` `/` `%` `mod`	`mod` is a keyword form of `%`
Additive	`+` `-`	overloaded for `Dist<T>` convolution
Comparison	`==` `=!` `<` `<=` `>` `>=`	
Approximate equality	`~=`	optionally followed by `within` *tolerance*
Logical	`&&` `||`	
Assignment / binding	`=`	used in `let` and in plain assignment

Table 2: Operator tokens.

Statement terminator

The semicolon `;` terminates every statement. Semicolons are *required*, not optional: an `if`-block used as a statement inside another block must itself end with a semicolon after its closing brace. Function and enum definitions at the top level are the only syntactic category that is not terminated by a semicolon.

2.6 Tokenisation

The tokeniser scans left to right with maximal munch: at each position it consumes the longest prefix that matches any token rule. Thus `<=` is a single comparison token, not `<` followed by `=`, and `->` is the function-arrow digraph, not `-` followed by `>`. Reserved keywords are matched with a trailing-character check: a keyword match succeeds only when the character immediately following is *not* a letter, digit, or underscore, so `mod_exp` is the single identifier `mod_exp` rather than the keyword `mod` followed by the identifier `_exp`.

After tokenisation, the resulting token stream is fed to the parser, whose grammar is given in full in §7.

3 Core Language Features

This section covers the deterministic core of YAPPL. A reader who knows any ML-family or Rust-like language can skim most of it: the surprises are concentrated in §4. The purpose of this section is to fix the syntactic forms that the probabilistic features are built on top of, and to make explicit what the language *deliberately omits*.

3.1 Programs

A YAPPL program is a sequence of top-level items. There are four kinds of top-level item: enum definitions, regular function definitions, probabilistic function definitions, and statements. Statements at the top level execute in order and share a single global scope; function and enum definitions are hoisted, so a function may call another defined later in the file.

Every statement ends in a semicolon. Function and enum definitions do not.

3.2 Bindings and Assignment

Variables are introduced with `let`:

```
let x = 42;
let name = "die"; // strings (see below)
let die = uniform(1, 6);
```

Re-assignment without `let` updates an existing binding:

```
x = x + 1;
```

A `let` with no initialiser declares a variable with a default value of zero (this is a concession to the imperative flavour of the prototype and is expected to be removed in favour of mandatory initialisation in a future revision).

3.3 Basic Types

YAPPL has four built-in scalar types and one type constructor for user-defined enums. The full hierarchy appears in §5; for now:

- `int`: 64-bit signed integers.
- `float`: IEEE-754 double-precision floats.
- `bool`: the two values `true` and `false`.
- `string`: finite sequences of Unicode code points. String literals are delimited by double quotes. Strings support equality and concatenation (via `+`).
- Named enum types (see §3.4).

Arithmetic, using the operators `+`, `-`, `*`, `/`, `%`, and `mod`, is defined on `int` and `float`, with the usual numeric-promotion rules when the two are mixed. The same `+` token is overloaded for distribution convolution (§4) and string concatenation; which meaning applies is determined by the types of the operands.

Comparison operators (`==`, `=!`, `<`, `<=`, `>`, `>=`) produce `bool`. Equality is defined pointwise on scalars, structurally on enums, and (§4) structurally on distributions. The approximate-equality operator `~=` is reserved for `Dist<T>` values and is described in §4.

Logical operators are `&&`, `||`, and prefix `!`. They are short-circuiting.

3.4 Enums (Sum Types)

A finite sum type is introduced with the `enum` keyword:

```
enum Weather { Sunny, Cloudy, Rainy }
```

Both the type name and each variant must begin with an uppercase letter (§2.2). Variants are values: once the enum is declared, `Sunny`, `Cloudy`, and `Rainy` can appear in expression position anywhere a `Weather` value is expected, and can be compared with `==` and `=!`:

```
if today == Sunny {
    ...
};
```

Enums are the language's primary tool for defining Markov-chain state spaces (§4); the parametric distribution type `Discrete<Weather>` uses an enum as its carrier.

Product types (records, tuples, structs) are *not* in the current language. The design intention is that any grouping of deterministic state that would be a record in another language is either a separate set of bindings, or, if it is the shape of a probabilistic outcome, a `Discrete<EnumType>` distribution whose outcomes cover the cases. This may be revisited.

3.5 Functions

Regular (deterministic) functions are introduced with `fn`:

```
fn square(n: int) -> int {
    return n * n;
}
```

```
fn max_of(a: int, b: int) -> int {
    if a > b { return a; };
    return b;
}
```

Parameters carry type annotations; the return type follows `->`. The body is a block; `return` produces a value. Recursion is allowed. Functions are *not* first-class values in the current language: a function name is a syntactic form, not a value that can be bound to a variable. The closest the language comes to treating functions as data is in the `bind`, `step`, and `map` forms (§4), which accept a *function name* as a syntactic argument. This is a deliberate simplification; a full first-class-function extension is a planned future revision.

3.6 Control Flow

The only control-flow construct in the deterministic core is the `if`/`else` statement:

```
if cond {
    ...
} else {
    ...
};
```

The `else` branch is optional. The `if` construct is a statement, not an expression, and its result is not a value; bindings made inside a branch are scoped to that branch.

Loops. YAPPL has no explicit looping construct. Iteration is expressed in one of three ways:

1. **Recursion** for general-purpose iteration.
2. `map` for applying a regular or probabilistic function pointwise over an array (§4).
3. `step` for iterating a Markov-chain transition function a fixed number of times (§4).

This omission is deliberate. A random-round loop would have to pick a repetition strategy and amplify its error in an ad-hoc way; that is exactly what `with confidence >=` automates at the call site. For deterministic counting loops, recursion or `map` covers the intended use cases.

Pattern matching. YAPPL does not (yet) have a general `match` form. Branching on an enum value is expressed by a chain of `if` comparisons:

```
fn weather_transition(today: Weather) -> Discrete<Weather> {
    if today == Sunny { return Discrete(Sunny: 0.7, ...); };
    if today == Cloudy { return Discrete(Cloudy: 0.7, ...); };
    return Discrete(Rainy: 0.7, ...);
}
```

A proper `match` form with exhaustiveness checking is a planned extension.

3.7 Arrays

Arrays are finite, ordered, heterogeneous sequences written with square-bracket literal syntax:

```
let primes = [2, 3, 5, 7, 11];
```

Arrays are the only collection type in the language. They exist primarily to feed `map`: the canonical use is a batch of primality tests or a batch of samples that share a confidence target. The `map` form treats the array length as the n in a union-bound split of the error budget (§4).

3.8 Output

The top-level `output` form prints a value, using a type-directed pretty printer:

```
output(42); // plain scalar
output(uniform(1, 6)); // distribution literal
output(die.visualise()); // histogram render
```

`output` is a language form, not a function, and is used only at the top level of a program. Output from inside a function body is not propagated to the top-level output buffer in the current implementation.

3.9 Modules

YAPPL does not yet have a module system. A program is a single source file. Reusable code is factored into functions and enum declarations within that file. A module system supporting separate compilation and namespacing is a planned future extension; its omission here is a limitation of the current language version, not a design choice to avoid the feature.

3.10 What this section deliberately leaves out

The deterministic core of YAPPL is consciously minimal. First-class functions, records, a module system, a general `match` form, and loops are all absent. The reason is that the language's purpose is to make the probabilistic parts (distributions as values, error classes in signatures, confidence-driven compilation) the centre of attention. Every feature omitted here is one that would have interacted with the error-class system in a way that had to be specified separately. Keeping the deterministic core small makes the probabilistic section, §4, the focus of the language.

4 Probabilistic Language Features

This section is the main content of the specification. It defines the `Dist<T>` type, its constructors and operations, the probabilistic function form `pb function`, the error-class system and the repetition strategies it drives, the confidence-target syntax and its distribution across multi-call blocks, the `distribution_of` operator that reifies the implicit distribution of a probabilistic function, the Bayesian extensions, and the Markov-chain primitives `bind` and `step`.

4.1 The `Dist<T>` Type

For every type `T` in the language, there is a type `Dist<T>` whose values are probability distributions over `T`. A `Dist<int>` is a distribution over integers; a `Dist<Weather>` is a distribution over the variants of an enum `Weather` and is written in source programs as `Discrete<Weather>`. The two spellings are interchangeable: `Discrete<T>` is the surface syntax that must appear in a type annotation, and `Dist<T>` is the metavariable used in typing rules and in prose.

A value of type `Dist<T>` is a first-class value in every sense: it may be bound, passed, returned, stored in an array, compared for equality, and operated on by the functions described in §4.3. Crucially, a distribution is *not* a sampling coroutine: it is a mathematical object that can be analysed *without* drawing any samples. Only the explicit `sample` method crosses from the pure algebraic view to the stochastic one.

4.2 Distribution Constructors

YAPPL provides seven built-in distribution constructors in the deterministic core of the language, plus two constructors that are introduced only as the result of `distribution_of` (§4.7) or Bayesian inference (§4.8). The full catalogue with types appears in §9; the core forms are:

Constructor	Result type	Meaning
`uniform(a, b)`	`Dist<int>`	discrete uniform on $\{a, a+1, \ldots, b\}$
`uniformContinuous(a, b)`	`Dist<float>`	continuous uniform on $[a, b)$
`Discrete(k_1: p_1, ...)`	`Dist<T>`	explicit PMF over keys of type `T`
`Bernoulli(p)`	`Dist<bool>`	$p = \Pr[\texttt{true}]$
`Binomial(n, p)`	`Dist<int>`	k successes in n Bernoulli(p) trials
`Geometric(p)`	`Dist<int>`	trials until first success, p per-trial
`Beta(alpha, beta)`	`Dist<float>`	conjugate prior/posterior on $[0, 1]$
`Point(x)`	`Dist<T>`	degenerate distribution at a single outcome

Table 3: Distribution constructors. `Point` is a degenerate case of `Discrete` (a single outcome with probability 1.0).

The `Discrete` form is the most general discrete constructor: it takes an explicit list of `key: probability` pairs, and the keys may be of any type that supports equality, including enum variants. This single constructor subsumes the role of a dedicated enum-valued discrete distribution: a `Discrete` over an enum type is a `Dist<E>` just as a `Discrete` over integers is a `Dist<int>`.

```
let fair_die = uniform(1, 6); // Dist<int>
let biased = Bernoulli(0.7); // Dist<bool>
let weather = Discrete(Sunny: 0.6, Cloudy: 0.3, Rainy: 0.1); // Dist<Weather>
```

4.3 Distribution Operations

Distributions support a small but algebraically complete set of operations. They fall into three groups: *combinators* that take one or more distributions and produce a new distribution, *queries* that inspect a distribution without sampling, and *observations* that draw samples.

4.3.1 Combinators

Sum (convolution). The `+` operator, when both operands are distributions, is the convolution: $(X + Y)$ is the distribution of the sum of independent samples from X and Y. This is computed analytically (no sampling) for discrete operands:

```
let two_dice = uniform(1, 6) + uniform(1, 6); // Dist<int> over 2..12
output(two_dice.expect(7)); // 1/6
```

Convolution is commutative, associative, and in the discrete case produces an exact result (stored as rational probabilities inside the runtime).

Monadic bind. The `bind` operator composes a distribution with a transition function $T \to$ `Dist<T>`:

$$\texttt{bind} \ : \ \texttt{Dist<T>} \times (T \to \texttt{Dist<T>}) \to \texttt{Dist<T>}$$

Given a distribution P over T and a function $f : T \to$ `Dist<T>`, `bind(P, f)` is the marginal distribution of the composite experiment "sample s from P, then sample from $f(s)$":

$$(\texttt{bind}(P, f))(s') \ = \ \sum_{s \in T} P(s) \cdot f(s)(s').$$

`bind` is the monadic bind for distributions and is computed analytically; no samples are drawn. It is the primary tool for propagating a distribution through one step of a Markov chain (§4.9).

Product (independent pairing). The Cartesian product of two distributions is written `product(d1, d2)` and has type `Dist<(T1, T2)>`. It is reserved for a future revision of the language: in the current core, joint behaviour is expressed either by convolution (when the observable is a sum) or by `bind` (when one distribution is conditioned on the other).

Map (pushforward). `d.map(f)` applies a pure function $f : T \to U$ pointwise to every outcome of `d` and returns a `Dist<U>`. It is not the same operator as the top-level `map` statement (which is an array-batched call form); the name is reused because both are functorial. The two are disambiguated by position: `d.map(f)` is a method call, `map(f, arr)` is a statement.

4.3.2 Queries

`expect(k)` / `prob(k)`. `d.expect(k)` returns the probability mass at outcome `k`, computed analytically. It is the core query for discrete distributions and is reported as an exact rational (type: `Fraction`, printed in its reduced form):

```
output(uniform(1, 6).expect(3)); // 1/6
output((uniform(1,6) + uniform(1,6)).expect(7)); // 1/6
```

For continuous distributions, `expect` is undefined and raises a static error in the type checker.

`mean()`. `d.mean()` returns the analytical expected value of `d`, as a `float` (or a `Fraction` where the exact answer is rational). It is defined on all distribution types for which a closed-form expectation exists, including the continuous cases `UniformContinuous` and `Beta`.

`entropy()`. `d.entropy()` returns Shannon entropy in nats. Reserved for a future revision; not in the prototype.

`min()`, `max()`. For uniform distributions only, `d.min()` and `d.max()` return the lower and upper bounds of the support.

`visualise()`. `d.visualise()` produces a rendering of the distribution suitable for `output`: an ASCII histogram in the CLI, an inline SVG in the web playground. For continuous distributions a small number of moments is reported in place of a full plot; for discrete distributions every outcome is shown.

4.3.3 Observations

`sample()`. `d:sample()` is the only primitive in the language that does something probabilistic: it takes a `Dist<T>` and returns a `T` drawn from d. Every other operation in §4.3 is purely analytical and produces no random output. This separation is deliberate: analytical reasoning about a program's distributions is possible exactly up to the point where `sample` is called.

4.3.4 Equality

Distribution values support both exact and approximate equality:

- `d1 == d2` is *structural* equality: the two distributions must have the same constructor and (recursively) equal parameters.
- `d1 ~= d2` is *approximate* equality. For two discrete distributions it tests that the total variation distance is at most a tolerance; for two continuous distributions it tests that the means and standard deviations agree within the tolerance. The default tolerance is 0.05; a

programmer-specified tolerance is written `d1 ~= d2 within t`, where `t` is any expression of type `float`.

Both forms are fully analytical; neither draws samples.

4.4 Probabilistic Functions

A *probabilistic function*, introduced with `pb function`, is a function whose single execution is one *round* of a repeatable experiment. A round returns either `Certain(v)` or `Uncertain(v)`, where `v` is a value of the function's declared return type. The signature declares two pieces of metadata that together constitute the probabilistic type of the function:

```
pb function is_prime(n: int) -> bool {
    error_class: RP,
    error_distribution: Geometric
} {
    // body: one round
}
```

The `error_class` declaration. The error class classifies the statistical guarantees of a single round. YAPPL supports three classes, following the classical randomised-complexity taxonomy:

Class	Formal meaning	Direction
`RP`	one-sided: `Certain(false)` is always correct	**no** is trusted
`coRP`	one-sided: `Certain(true)` is always correct	**yes** is trusted
`BPP`	two-sided: both answers may be wrong with bounded bias	majority vote

Table 4: Error classes. Conceptually, `RP` and `coRP` are the "one-sided with direction" case and `BPP` is the "two-sided" case. A hypothetical fourth class `certain` would correspond to a deterministic function and is simply a regular `fn`.

The error class is part of the function's type: two `pb function` values are not interchangeable unless they have the same signature *including* the error class. This is what the typing rules in §5 encode.

The `error_distribution` declaration. The error-distribution field names a distribution family describing how the per-round error decays with repetition. In the prototype this is informational (the runtime derives the round count from the error class directly), but it serves as machine-readable documentation: a `Geometric` error distribution says that each `Uncertain` round halves the residual error, while a `Binomial` error distribution says that the round is a fair coin toward the correct answer and amplification must use a majority vote.

Round bodies. The body of a `pb function` must terminate every path with either `return Certain(v)` or `return Uncertain(v)`. A return of a bare value is a type error. Regular (deterministic) control flow, distribution sampling, and calls to regular functions are all allowed inside a round. Calling another `pb function` from inside a round is *not* allowed: all composition of probabilistic functions happens at the call site, via the confidence mechanisms of §4.5.

4.5 Confidence Targets and Repetition Strategies

A probabilistic function is never called the way a regular function is. Instead, the caller declares a *confidence target* and lets the compiler pick the number of rounds:

```
let result, info = is_prime(53) with confidence >= 0.99;
```

The call binds two variables: `result` holds the final answer, and `info` holds a record `Info { rounds , confidence }` containing the number of rounds actually run and the actual confidence achieved (which is always at least the requested target). The target c must be a `float` literal in $[0, 1)$.

The compiler translates the target into a repetition strategy determined entirely by the function's error class:

RP (short-circuit on `Certain`). Run rounds in sequence. On the first `Certain(v)`, return v with confidence 1.0 (no residual error). If all k rounds return `Uncertain(v)`, the combined error is bounded by $(1/2)^k$, giving a confidence of $1 - (1/2)^k$. The smallest k that achieves a target c is

$$k_{\mathrm{RP}}(c) \;=\; \lceil -\log_2(1-c) \rceil.$$

E.g. $c = 0.99 \Rightarrow k = 7$; $c = 0.999 \Rightarrow k = 10$.

coRP (dual of RP). Identical repetition strategy to `RP` with the roles of "certain" and "uncertain" swapped: a `Certain(true)` short-circuits, and all `Uncertain(false)` rounds accumulate toward confidence $1 - (1/2)^k$.

BPP (majority vote with Chernoff bound). Run all k rounds to completion, tally the outcomes, and return the majority. Assuming a per-round success probability of $3/4$ (the standard BPP amplification assumption), the Chernoff bound gives an error probability of at most $\exp(-k/8)$, so the round count is

$$k_{\mathrm{BPP}}(c) \;=\; \lceil -8 \ln(1-c) \rceil.$$

E.g. $c = 0.99 \Rightarrow k = 37$.

The programmer does not write any of these formulas: they live in the compiler, keyed on the error class. The only thing a call site says is the target.

4.6 Block-Level Confidence and Bonferroni Distribution

A confidence target attached to a *single* probabilistic call governs only that call. When a program makes several probabilistic calls whose answers must *all* be correct simultaneously, the error budget must be distributed across them. YAPPL does this via a union bound (Bonferroni correction).

The `map` form. The simplest case is applying a `pb function` to every element of an array:

```
let primes = map(is_prime, [53, 97, 101, 7919])
           with confidence >= 0.99;
```

The runtime is required to return an answer array in which *every* element is correct with overall confidence at least 0.99. By the union bound, it is sufficient to achieve per-element confidence $1 - (1 - 0.99)/n$ where n is the array length, so the per-element target is $1 - 0.01/4 = 0.9975$ and each call runs with that tightened target.

Multi-call confidence blocks (specification extension). For general composition of several probabilistic calls in the same block, the specification describes a syntactic form that distributes the error budget across them:

```
with confidence >= 0.99 {
    let r1, _ = is_prime(p) with confidence;
    let r2, _ = is_prime(q) with confidence;
    let r3, _ = passes_test(x) with confidence;
    output(r1 && r2 && r3);
}
```

53

Inside the block, each `with confidence` (without a target) inherits a per-call target of $1-(1-c)/m$, where c is the block target and m is the number of probabilistic calls textually present in the block. The block form is reserved in the specification but is not yet parsed by the prototype; the `map` form is the concrete available illustration of Bonferroni distribution.

4.7 The `distribution_of` Operator

For any `pb function f`, the operator `distribution_of` reifies the implicit per-round distribution of `f` into a first-class `Dist` value. It takes three forms, each answering a different question:

```
let d1 = distribution_of(is_prime(53), analytical);
let d2 = distribution_of(is_prime(53), empirical, 1000);
let d3 = distribution_of(is_prime(53), bayesian, 1000);
```

> Produces the distribution implied by the declared error class of `f`. For `RP`, the distribution is `Geometric(0.5)`: the round index at which the first `Certain` answer is expected to appear under the worst-case per-round error bound. For `BPP`, it is `Bernoulli(0.75)`: the per-round success probability assumed by the Chernoff analysis. No sampling is performed.
>
> Runs `f` for N single rounds (no repetition), tallies how many return `Certain` vs `Uncertain`, and returns a `Bernoulli(p)` where p is the empirical fraction. Default $N = 100$. This is a frequentist point estimate of the per-round success probability.
>
> Runs `f` for N single rounds and constructs a Beta posterior over the per-round success probability, starting from a uniform prior `Beta(1, 1)`. With c certains and u uncertains, the result is `Beta(1 + c, 1 + u)`. Default $N = 100$.

Applied and curried forms. The form shown above is the *applied* form: it names the arguments of `f` and returns the per-round distribution of that specific call. A *curried* form is reserved in the specification:

```
let f_dist = distribution_of(is_prime, analytical); // Int -> Dist<bool>
let d = f_dist(53);
```

The curried form produces a function that, given the same arguments `f` would take, returns a distribution. It is not in the prototype; the applied form is the available one.

4.8 Bayesian Extensions

The Bayesian mode of `distribution_of` above is the simplest entry point to the Bayesian extension. The full extension (reserved for a future revision of the language) adds:

Priors. A `prior` keyword marks a distribution value as a prior belief about a parameter:

```
let prior bias = Beta(1, 1); // uninformative prior over a coin bias
```

Posteriors. Given a prior and observations, the `posterior` operator returns the Bayesian posterior over the parameter. In the prototype, this is implemented specifically for `Bernoulli` observations with a `Beta` prior (via the `bayesian` mode of `distribution_of`); the full specification generalises this to arbitrary conjugate-prior families.

The `with bayesian_confidence` target. Whereas `with confidence >= c` interprets c as a frequentist lower bound on the error probability, an alternative target form interprets c as a Bayesian posterior probability:

```
let result, info = is_prime(n) with bayesian_confidence >= 0.99;
```

Under `bayesian_confidence`, the runtime computes the posterior over the per-round success probability (using the declared error distribution as its conjugate family) and continues rounds until the posterior probability of the correct answer exceeds c. This is a proper extension of the frequentist form and is reserved; the prototype implements only the frequentist target.

4.9 Markov Chain Primitives

A Markov chain over a finite state space is expressed by an enum declaring the states and a regular function that gives the transition distribution from each state:

```
enum Weather { Cloudy, Rainy, Sunny }

fn weather_transition(today: Weather) -> Discrete<Weather> {
    if today == Sunny { return Discrete(Sunny: 0.7, Cloudy: 0.2, Rainy: 0.1); };
    if today == Cloudy { return Discrete(Cloudy: 0.7, Sunny: 0.2, Rainy: 0.1); };
    return Discrete(Rainy: 0.7, Cloudy: 0.2, Sunny: 0.1);
}
```

Note that this is a *regular* `fn`, not a `pb function`: its behaviour is deterministic in the mathematical sense (given the same input, it returns the same distribution). The stochastic content is in the distribution it returns.

Two primitives operate on transition functions.

`bind(dist, f)`. Given a distribution `dist: Dist<T>` and a transition function `f: T -> Dist<T>`, `bind(dist, f)` is the one-step image of `dist` under `f`, computed analytically by marginalising:

$$(\texttt{bind}(P, f))(s') \;=\; \sum_{s} P(s) \cdot f(s)(s').$$

This is the same monadic bind described in §4.3, specialised to the endomorphic case $T \to \texttt{Dist<T>}$ that characterises Markov transitions.

`step(initial, f, n)`. Given an initial state `initial: T`, a transition function `f: T -> Dist<T>`, and a non-negative integer `n`, `step` returns the marginal distribution over states after n applications of `f`, starting from the delta distribution concentrated at `initial`:

$$\texttt{step}(s_0, f, n) \;=\; \underbrace{f \circ \cdots \circ f}_{n}\,[\,\texttt{Point}(s_0)\,].$$

For large n, `step` converges to the stationary distribution (when one exists). Example:

```
let stationary = step(Sunny, weather_transition, 50);
output(stationary.visualise());
```

Argument form. Both `bind` and `step` take a *function name* as one of their arguments, not a function value: this is a consequence of regular functions not being first-class in the current language (§3.5). When first-class functions are added, both forms will accept function-valued expressions as well.

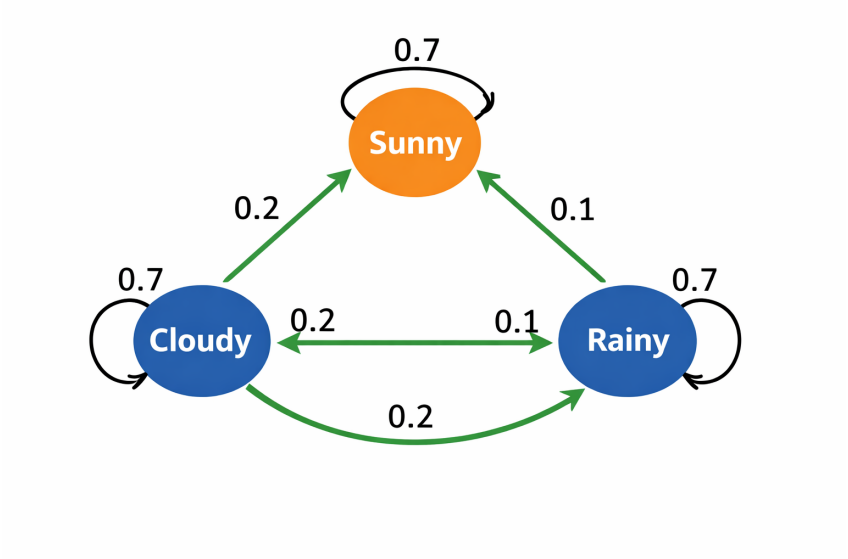

Figure 1: The weather Markov chain as a state-transition diagram. Each arrow is annotated with the per-step probability of moving from one weather state to another; `step` iterates this transition function and converges the initial delta distribution toward a fixed point (the stationary distribution).

5 Type System

This section formalises the YAPPL type system. The main points of interest are (a) the parametric `Dist<T>` type and how its operations are typed, (b) the treatment of error classes as part of a `pb function`'s function type, and (c) the typing rule for `distribution_of`, which is the formal link between the `pb function` world and the `Dist` world.

5.1 Type Grammar

The abstract type syntax is:

$$
\begin{array}{rcl}
\tau & ::= & \texttt{int} \mid \texttt{float} \mid \texttt{bool} \mid \texttt{string} \\
& \mid & E \quad \text{(a named enum type)} \\
& \mid & \texttt{Dist}\langle \tau \rangle \\
& \mid & [\tau] \quad \text{(array of } \tau\text{)} \\
\varphi & ::= & \tau_1, \ldots, \tau_n \to \tau \quad \text{(regular function type)} \\
\pi & ::= & \tau_1, \ldots, \tau_n \to^{\kappa} \tau \quad \text{(probabilistic function type)} \\
\kappa & ::= & \texttt{RP} \mid \texttt{coRP} \mid \texttt{BPP}
\end{array}
$$

Here τ ranges over *value types*, φ over regular function types, π over probabilistic function types, and κ over error classes. A probabilistic function type is annotated on its arrow with its error class; two `pb function`s with the same parameter types, same return type, but different error classes are *different types* and are not interchangeable.

A typing context Γ maps identifiers to value types or function types. The typing judgement for expressions is $\Gamma \vdash e : \tau$; the judgement for statements is $\Gamma \vdash s \Rightarrow \Gamma'$, meaning that the statement s is well-typed in Γ and extends the context to Γ'.

5.2 Typing Rules for Core Constructs

The rules for the deterministic core are conventional and listed here for completeness. Literals give their respective types:

$$\frac{}{\Gamma \vdash n : \texttt{int}}\,(\text{T-Int}) \qquad \frac{}{\Gamma \vdash f : \texttt{float}}\,(\text{T-Float}) \qquad \frac{}{\Gamma \vdash b : \texttt{bool}}\,(\text{T-Bool})$$

Arithmetic is defined on `int` and `float` and promotes mixed operands to `float`:

$$\frac{\Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau \qquad \tau \in \{\texttt{int}, \texttt{float}\}}{\Gamma \vdash e_1 + e_2 : \tau}\,(\text{T-Add})$$

Comparisons produce `bool`; logical operators require `bool` operands. Regular function application uses the standard rule:

$$\frac{\Gamma(f) = \tau_1, \ldots, \tau_n \to \tau \qquad \Gamma \vdash e_i : \tau_i \text{ for each } i}{\Gamma \vdash f(e_1, \ldots, e_n) : \tau}\,(\text{T-Call})$$

5.3 Typing Rules for Distributions

Distribution constructors each have a fixed signature. A selection:

$$\frac{\Gamma \vdash a : \texttt{int} \qquad \Gamma \vdash b : \texttt{int}}{\Gamma \vdash \texttt{uniform}(a, b) : \texttt{Dist}\langle\texttt{int}\rangle}\,(\text{T-Uniform})$$

$$\frac{\Gamma \vdash p : \texttt{float}}{\Gamma \vdash \texttt{Bernoulli}(p) : \texttt{Dist}\langle\texttt{bool}\rangle}\,(\text{T-Bernoulli})$$

$$\frac{\Gamma \vdash n : \texttt{int} \qquad \Gamma \vdash p : \texttt{float}}{\Gamma \vdash \texttt{Binomial}(n, p) : \texttt{Dist}\langle\texttt{int}\rangle}\,(\text{T-Binomial})$$

$$\frac{\Gamma \vdash p : \texttt{float}}{\Gamma \vdash \texttt{Geometric}(p) : \texttt{Dist}\langle\texttt{int}\rangle}\,(\text{T-Geometric})$$

The rule for `Discrete` infers the element type from the keys:

$$\frac{\begin{array}{c}\Gamma \vdash k_i : \tau \text{ for each } i \\ \Gamma \vdash p_i : \texttt{float} \text{ for each } i\end{array}}{\Gamma \vdash \texttt{Discrete}(k_1 : p_1, \ldots, k_n : p_n) : \texttt{Dist}\langle\tau\rangle}\,(\text{T-Discrete})$$

Sum (convolution) is typed on two distributions of the same numeric carrier:

$$\frac{\Gamma \vdash d_1 : \texttt{Dist}\langle\tau\rangle \qquad \Gamma \vdash d_2 : \texttt{Dist}\langle\tau\rangle \qquad \tau \in \{\texttt{int}, \texttt{float}\}}{\Gamma \vdash d_1 + d_2 : \texttt{Dist}\langle\tau\rangle}\,(\text{T-DSum})$$

The methods on distributions are typed as follows:

$$\frac{\Gamma \vdash d : \texttt{Dist}\langle\tau\rangle}{\Gamma \vdash d.\texttt{sample}() : \tau}\,(\text{T-Sample}) \qquad \frac{\Gamma \vdash d : \texttt{Dist}\langle\tau\rangle \qquad \tau \text{ discrete}}{\Gamma \vdash d.\texttt{expect}(k) : \texttt{float}}\,(\text{T-Expect})$$

$$\frac{\Gamma \vdash d : \texttt{Dist}\langle\tau\rangle}{\Gamma \vdash d.\texttt{mean}() : \texttt{float}}\,(\text{T-Mean})$$

Approximate equality requires both sides to be distributions; an optional tolerance must be a `float`:

$$\frac{\Gamma \vdash d_1 : \texttt{Dist}\langle\tau\rangle \qquad \Gamma \vdash d_2 : \texttt{Dist}\langle\tau\rangle \qquad \Gamma \vdash t : \texttt{float}}{\Gamma \vdash d_1 \eqsim d_2 \texttt{ within } t : \texttt{bool}}\,(\text{T-ApproxEq})$$

5.4 Typing Rules for Probabilistic Functions

A `pb function` definition introduces a binding of probabilistic function type π into the global context. The definition is well-typed if its body, extended with the parameter bindings, returns `Certain(v)` or `Uncertain(v)` where `v` has the declared return type:

$$\frac{\Gamma, x_1 : \tau_1, \ldots, x_n : \tau_n \vdash \mathit{body} : \texttt{Cert}\langle\tau\rangle}{\Gamma \vdash \texttt{pb function}\ f(x_1 : \tau_1, \ldots, x_n : \tau_n) \to^{\kappa} \tau\ \{\ldots\}\ \mathit{body}\ \textbf{ok}}\ (\text{T-PbFnDef})$$

where $\texttt{Cert}\langle\tau\rangle$ is an internal marker type for a body that always returns either `Certain(v)` or `Uncertain(v)` for some `v : $\tau$`. The certainty-wrapping rules are:

$$\frac{\Gamma \vdash v : \tau}{\Gamma \vdash \texttt{Certain}(v) : \texttt{Cert}\langle\tau\rangle}\ (\text{T-Certain}) \qquad \frac{\Gamma \vdash v : \tau}{\Gamma \vdash \texttt{Uncertain}(v) : \texttt{Cert}\langle\tau\rangle}\ (\text{T-Uncertain})$$

The `with confidence` call form is a statement that binds a result variable and an info variable:

$$\frac{\begin{array}{c}\Gamma(f) = \tau_1, \ldots, \tau_n \to^{\kappa} \tau \\ \Gamma \vdash e_i : \tau_i \text{ for each } i \\ c \in [0, 1)\end{array}}{\Gamma \vdash \texttt{let}\ r, i = f(e_1, \ldots, e_n)\ \texttt{with confidence} \geq c \quad \Rightarrow\ \Gamma, r : \tau, i : \texttt{Info}}\ (\text{T-PbCall})$$

Here the error class κ is not used by the *typing* judgement but is consulted by the operational semantics (§8) to pick the repetition strategy. Two `pb function`s with identical τ_i, τ but different κ are still different types: an implementation that caches or dispatches on function type must preserve the distinction.

5.5 Typing Rule for `distribution_of`

The key rule linking the two worlds. The *applied* form takes a fully applied probabilistic call and returns a distribution over the return type of that call:

$$\frac{\begin{array}{c}\Gamma(f) = \tau_1, \ldots, \tau_n \to^{\kappa} \tau \\ \Gamma \vdash e_i : \tau_i \text{ for each } i \\ m \in \{\texttt{analytical}, \texttt{empirical}(N), \texttt{bayesian}(N)\}\end{array}}{\Gamma \vdash \texttt{distribution\_of}(f(e_1, \ldots, e_n), m) : \texttt{Dist}\langle\tau\rangle}\ (\text{T-DistOf})$$

The *curried* form (reserved) has a higher-order type that "strips" the error class from the arrow:

$$\frac{\Gamma(f) = \tau_1, \ldots, \tau_n \to^{\kappa} \tau}{\Gamma \vdash \texttt{distribution\_of}(f, \texttt{analytical}) : \tau_1, \ldots, \tau_n \to \texttt{Dist}\langle\tau\rangle}\ (\text{T-DistOfCurried})$$

This rule is the type-level statement of the duality pillar of §4: every `pb function` of type $\tau_1, \ldots, \tau_n \to^{\kappa} \tau$ has a shadow regular function of type $\tau_1, \ldots, \tau_n \to \texttt{Dist}\langle\tau\rangle$. The error class κ is *erased* by `distribution_of`; once a function has been reified into a distribution, there is no further amplification to do, and the `with confidence` machinery no longer applies.

5.6 Typing Rules for Markov Primitives

The Markov primitives `bind` and `step` take a transition function as an argument:

$$\frac{\begin{array}{c}\Gamma \vdash d : \texttt{Dist}\langle\tau\rangle \\ \Gamma(f) = \tau \to \texttt{Dist}\langle\tau\rangle\end{array}}{\Gamma \vdash \texttt{bind}(d, f) : \texttt{Dist}\langle\tau\rangle}\ (\text{T-Bind})$$

$$\frac{\begin{array}{c}\Gamma \vdash s_0 : \tau \\ \Gamma(f) = \tau \to \texttt{Dist}\langle\tau\rangle \\ \Gamma \vdash n : \texttt{int}\end{array}}{\Gamma \vdash \texttt{step}(s_0, f, n) : \texttt{Dist}\langle\tau\rangle}\,(\text{T-Step})$$

Note that f must be a regular transition function, not a `pb function`: the Markov primitives do not live in the error-classified world.

5.7 Typing Rule for Union-Bound `map`

The `map` statement form with a confidence target takes a probabilistic function and an array, and returns an array of results such that every element is correct with confidence at least c simultaneously:

$$\frac{\begin{array}{c}\Gamma(f) = \tau \to^{\kappa} \tau' \\ \Gamma \vdash a : [\tau] \\ c \in [0, 1)\end{array}}{\Gamma \vdash \texttt{let } v = \texttt{map}(f, a) \texttt{ with confidence} \geq c \;\Rightarrow\; \Gamma, v : [\tau']}\,(\text{T-MapConf})$$

The per-element target used by the runtime (Bonferroni correction) is derived *operationally* from c and the length of a; the typing rule only records that the overall target is c.

5.8 Typing Judgement for Approximate Equality with Default Tolerance

The tolerance expression in `d1 ~= d2 within t` is optional in the source syntax; the typing rule with the default tolerance is:

$$\frac{\Gamma \vdash d_1 : \texttt{Dist}\langle\tau\rangle \qquad \Gamma \vdash d_2 : \texttt{Dist}\langle\tau\rangle}{\Gamma \vdash d_1 \mathrel{\tilde{=}} d_2 : \texttt{bool}}\,(\text{T-ApproxEqDefault})$$

The default tolerance is 0.05; see §4.3.

5.9 Type Safety

The usual progress and preservation theorems hold for the deterministic core in the standard way. For the probabilistic fragment, the statement of preservation is modified to account for the fact that one rewrite step happens at the level of distributions: if $\Gamma \vdash e : \tau$ and $e \longrightarrow e'$, then $\Gamma \vdash e' : \tau$ still holds, where the rewrite relation $\longrightarrow$ is the small-step distribution-based relation defined in §8. The key lemma is that `distribution_of` preserves the return type and that `with confidence` preserves the return type while discarding the error class.

6 Examples

This section walks through four worked examples that together exercise every major probabilistic feature in the language: a one-sided RP primality test, a multi-call composition illustrating Bonferroni distribution, a Markov-chain weather model with a stationary- distribution query, and a use of `distribution_of` to inspect the implicit distribution of a probabilistic function. The commentary identifies which section introduced each feature used.

6.1 Example 1: Solovay–Strassen Primality (RP)

The Solovay–Strassen primality test is the canonical example of an `RP` algorithm: a single round is a coin flip with one-sided error. A composite n is detected as composite with probability at least 1/2 (a `Certain(false)` answer is always correct); a prime n always returns `Uncertain(true)`.[59]

```
pb function is_prime(p: int) -> bool {
    error_class: RP,
    error_distribution: Geometric
} {
    if p < 2 { return Certain(false); };
    if p == 2 { return Certain(true); };
    if p % 2 == 0 { return Certain(false); };

    a = uniform(1, p - 1).sample();
    jacobian = (p + jacobi(a, p)) % p;
    euler = mod_exp(a, (p - 1) / 2, p);

    if jacobian == 0 { return Certain(false); };
    if euler != jacobian { return Certain(false); };

    return Uncertain(true);
}

let result, info = is_prime(53) with confidence >= 0.99;
output(result); // true
output(info); // Info { rounds: 7, confidence: 0.992188 }
```

What the example demonstrates.

analytical empirical Novel pb function form and the two-field metadata block (§4.4).

- Certainty markers `Certain(v)` and `Uncertain(v)` as the only legal return forms of a round body.
- Sampling a distribution (`uniform(1, p-1).sample()`) inside a round body.
- The confidence-target call form (§4.5) and the derived round count $k_{\mathrm{RP}}(0.99) = \lceil -\log_2 0.01 \rceil = 7$.
- The `info` record reporting the actual rounds-and-confidence.

6.2 Example 2: Miller–Rabin and Multi-Call Composition

Miller–Rabin is a second `RP` primality test, independent of Solovay–Strassen. We use it here to illustrate *composition*: the programmer wants both tests to agree on a prime, and the overall error budget must be distributed across the two calls via the union bound (§4.6).

```
pb function miller_rabin(p: int) -> bool {
    error_class: RP,
    error_distribution: Geometric
} {
    if p < 2 { return Certain(false); };
    if p == 2 { return Certain(true); };
    if p % 2 == 0 { return Certain(false); };

    // One round of Miller-Rabin: pick a witness, do the strong-test
    // (body elided; uses mod_exp, bit-decomposition of p-1, etc.)
    // ...
    return Uncertain(true);
}

// The confidence-block form: both calls must succeed simultaneously
// at confidence >= 0.999. The compiler tightens each per-call target
// to 1 - (1 - 0.999) / 2 = 0.9995 and runs them independently.
with confidence >= 0.999 {
    let r1, _ = is_prime(p_candidate) with confidence;
```

```
    let r2, _ = miller_rabin(p_candidate) with confidence;
    output(r1 && r2);
}
```

What the example demonstrates.

- A second independent `pb function` with the same error class.
- The `with confidence >= c { ... }` block form (§4.6): the outer target is 0.999; each inner `with confidence` (bare, no target) inherits a per-call target of $1 - (1 - 0.999)/2 = 0.9995$.
- Bonferroni distribution in action: the overall probability that *at least one* of the two calls returns a wrong answer is at most $(1 - 0.9995) + (1 - 0.9995) = 0.001$, so the conjunction is correct with probability at least 0.999, as targeted.

The block form is reserved in the specification; see §4.6 for its status in the prototype (the `map` form in Example 4 is the analogue that *is* implemented).

6.3 Example 3: Markov Chain Weather Model

A finite-state Markov chain is expressed with an `enum` for the state type, a regular function for the transition, and `step` to compute the distribution after n applications.

```
enum Weather { Cloudy, Rainy, Sunny }

fn weather_transition(today: Weather) -> Discrete<Weather> {
    if today == Sunny { return Discrete(Sunny: 0.7, Cloudy: 0.2, Rainy: 0.1); };
    if today == Cloudy { return Discrete(Cloudy: 0.7, Sunny: 0.2, Rainy: 0.1); };
    return Discrete(Rainy: 0.7, Cloudy: 0.2, Sunny: 0.1);
}

// One step from a deterministic initial state.
let after_1 = bind(Discrete(Sunny: 1.0), weather_transition);
output(after_1);

// 50 steps: should be close to the stationary distribution.
let stationary = step(Sunny, weather_transition, 50);
output(stationary.visualise());
```

Running the program gives something close to $(P(\text{Cloudy}), P(\text{Rainy}), P(\text{Sunny})) = (0.40, 0.25, 0.35)$, which is the stationary distribution of this transition matrix.

What the example demonstrates.

- User-defined enum types as Markov-chain state spaces (§3.4).
- A regular `fn` returning a `Discrete<Weather>`: a transition function is a pure function that *returns* a distribution.
- `bind` as the monadic bind for distributions, threaded with the transition function with a transition function.
- `step` as repeated `bind`, converging to the stationary distribution (§4.9).
- `visualise()` as a method on distribution values (§4.3).

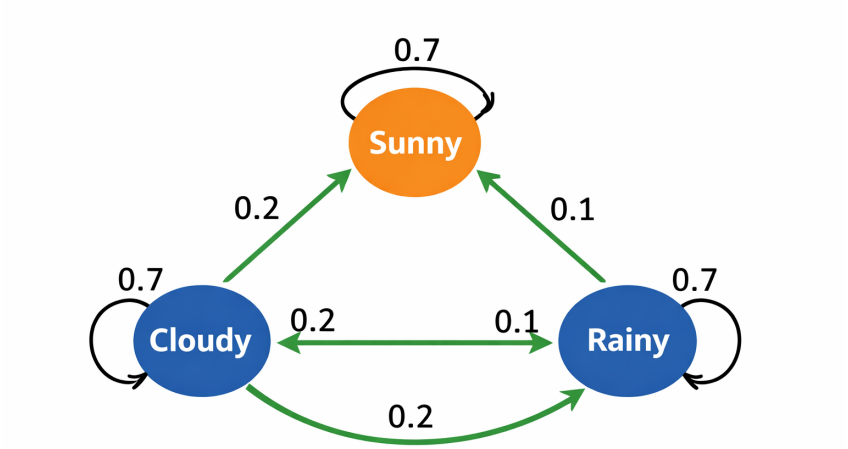

Figure 2: The same weather chain as a state-transition diagram, repeated here for the worked example.

6.4 Example 4: Inspecting a Probabilistic Function with `distribution_of`

The `distribution_of` operator reifies the per-round behaviour of a probabilistic function into a first-class distribution value, so the programmer can analyse it without running `with confidence` amplification.

```
// Analytical: the distribution implied by the error class metadata.
// For RP, this is Geometric(0.5).
let d_analytical = distribution_of(is_prime(53), analytical);
output(d_analytical); // Geometric(0.5)

// Empirical: run 1000 single rounds and report the Certain fraction.
let d_empirical = distribution_of(is_prime(53), empirical, 1000);
output(d_empirical); // Bernoulli(p) with p empirical

// Bayesian: Beta posterior over the per-round Certain probability
// with a Beta(1,1) prior and 1000 observations.
let d_bayesian = distribution_of(is_prime(53), bayesian, 1000);
output(d_bayesian.visualise()); // Beta(alpha, beta)

// Compare the empirical and analytical distributions approximately.
if d_empirical ~= Bernoulli(1.0) within 0.05 {
    output("is_prime(53) almost never returns Certain(false) -- good!");
};
```

What the example demonstrates.

- All three modes of `distribution_of` (§4.7): analytical (no sampling), empirical (point estimate), and Bayesian (posterior).
- The result is an ordinary first-class `Dist` value that can be stored, printed, visualised, and compared.
- Approximate equality `~=` with a tolerance, used to test whether an empirical distribution matches an expected analytical one (§4.3).

Together, Examples 1–4 cover every major construct in §4: probabilistic function definition, error classes, confidence targets (single-call and multi-call), Markov chain primitives, and

distribution_of.

7 Formal Syntax (BNF)

This section gives the full concrete grammar of YAPPL in BNF. The grammar below is also the canonical reference grammar distributed with the compiler (as `grammar.bnf`) and is kept in sync with the parser. Non-terminals are written in lowercase, terminals in double quotes. The metacharacters `?`, `*`, and `+` denote zero-or-one, zero-or-more, and one-or-more repetitions; parentheses group; `|` is alternation; and `epsilon` is the empty production.

An abbreviated abstract syntax for use in the operational semantics of §8 follows after the concrete grammar.

7.1 Concrete Grammar

```
# Top-Level

program ::= program_item*
program_item ::= enum_def
                | fn_def
                | pb_fn_def
                | statement ";"

# Enum Definitions

enum_def ::= "enum" UIDENT "{" variant_list "}"
variant_list ::= UIDENT ("," UIDENT)*

# Function Definitions

fn_def ::= "fn" IDENT "(" param_list ")" "->" type block
pb_fn_def ::= "pb" "function" IDENT "(" param_list ")" "->" type
                  "{" pb_metadata "}" block

pb_metadata ::= "error_class" ":" error_class ","
                  "error_distribution" ":" IDENT
error_class ::= "RP" | "coRP" | "BPP"

param_list ::= param ("," param)* | epsilon
param ::= IDENT ":" type

type ::= "int" | "float" | "bool"
                | "Discrete" "<" UIDENT ">"
                | UIDENT

# Blocks and Statements

block ::= "{" statement_list "}"
statement_list ::= (statement ";")*

statement ::= pb_call_assign
                | distribution_of_stmt
                | map_call_assign
                | return_stmt
                | if_stmt
                | var_declaration
                | hardcoded_output
                | statement_assignment

pb_call_assign ::= "let" IDENT "," IDENT "=" IDENT "(" arg_list ")"
                  "with" "confidence" ">=" FLOAT_LIT

distribution_of_stmt ::=
     "let" IDENT "=" "distribution_of" "(" IDENT "(" arg_list ")" "," of_mode ")"
of_mode ::= "analytical"
                | "empirical" ("," INT_LIT)?
                | "bayesian" ("," INT_LIT)?
```

63

```
map_call_assign ::= "let" IDENT "=" "map" "(" IDENT "," expr ")"
                  ("with" "confidence" ">=" FLOAT_LIT)?

return_stmt ::= "return" expr?
if_stmt ::= "if" expr block ("else" block)?
var_declaration ::= "let" IDENT ("=" expr)?
statement_assignment ::= IDENT "=" expr
hardcoded_output ::= "output(" expr ")"

# Expressions
# Precedence (lowest to highest):
# OR -> AND -> CMP -> ADD -> MUL -> UNARY -> PRIMARY

expr ::= or_expr
or_expr ::= and_expr ("||" and_expr)*
and_expr ::= cmp_expr ("&&" cmp_expr)*

cmp_expr ::= add_expr (cmp_tail)?
cmp_tail ::= "~=" add_expr ("within" add_expr)?
                | cmp_op add_expr
cmp_op ::= "==" | "!=" | "<=" | ">=" | "<" | ">"

add_expr ::= mul_term (("+"|"-") mul_term)*
mul_term ::= unary_term (("*"|"/"|"%"|"mod") unary_term)*
unary_term ::= "-" unary_term
                | "!" unary_term
                | primary_postfix

primary_postfix ::= primary_term method_call*
method_call ::= ("."|":") IDENT "(" arg_list ")"

primary_term ::= "(" expr ")"
                | array_literal
                | "true" | "false"
                | func_call
                | var_usage
                | NUMBER_LIT

array_literal ::= "[" arg_list "]"
func_call ::= IDENT "(" arg_list ")"
var_usage ::= IDENT
arg_list ::= expr ("," expr)* | epsilon

# Distribution Constructors (special func_call names)

dist_ctor ::= "uniform" "(" expr "," expr ")"
                | "uniformContinuous" "(" expr "," expr ")"
                | "Discrete" "(" discrete_pair_list ")"
                | "Bernoulli" "(" expr ")"
                | "Binomial" "(" expr "," expr ")"
                | "Geometric" "(" expr ")"

discrete_pair_list ::= discrete_pair ("," discrete_pair)*
discrete_pair ::= expr ":" expr

# Certainty Markers

certainty_call ::= "Certain" "(" expr ")"
                | "Uncertain" "(" expr ")"

# Markov-Chain Built-in Forms

markov_call ::= "bind" "(" expr "," IDENT ")"
                | "step" "(" expr "," IDENT "," expr ")"

# Literals

NUMBER_LIT ::= INT_LIT | FLOAT_LIT
INT_LIT ::= [0-9]+
FLOAT_LIT ::= [0-9]+ "." [0-9]+

IDENT ::= [a-zA-Z_][a-zA-Z0-9_]*
UIDENT ::= [A-Z][a-zA-Z0-9_]*
```

```
# Comments
COMMENT ::= "//" [^\n]* "\n"
```

7.2 Abstract Syntax

The operational semantics of §8 works over the following simplified abstract syntax. Desugaring from the concrete grammar is straightforward and mostly erases parenthesisation, operator precedence, and sugar for `let`-with-initialiser.

$$
\begin{array}{rcl}
v & ::= & n \mid f \mid b \mid \texttt{EnumVar}(E, V) \mid d \quad \text{(values)} \\
d & ::= & \texttt{Uniform}(v, v) \mid \texttt{Bernoulli}(v) \mid \texttt{Binomial}(v, v) \\
 & \mid & \texttt{Geometric}(v) \mid \texttt{Discrete}(\overline{v : v}) \\
 & \mid & \texttt{Beta}(v, v) \mid d \oplus d \quad \text{(distribution values)} \\
e & ::= & v \mid x \mid e \oplus e \mid e \otimes e \mid \ldots \quad \text{(arith/cmp elided)} \\
 & \mid & f(\overline{e}) \quad \text{(regular call)} \\
 & \mid & e.\texttt{sample}() \mid e.\texttt{expect}(e) \mid e.\texttt{mean}() \mid e.\texttt{visualise}() \\
 & \mid & \texttt{bind}(e, f) \mid \texttt{step}(e, f, e) \\
 & \mid & \texttt{Certain}(e) \mid \texttt{Uncertain}(e) \\
 & \mid & e \sim\!\!= e \ [\texttt{within}\ e] \\
s & ::= & \texttt{let}\ x = e \mid x := e \mid \texttt{return}\ e \\
 & \mid & \texttt{if}\ e\ \{\overline{s}\}\ [\texttt{else}\ \{\overline{s}\}] \\
 & \mid & \texttt{let}\ r, i = f(\overline{e})\ \texttt{with conf} \geq c \quad \text{(pb-call)} \\
 & \mid & \texttt{let}\ v = \texttt{distribution\_of}(f(\overline{e}), m) \\
 & \mid & \texttt{let}\ v = \texttt{map}(f, e)\ [\texttt{with conf} \geq c] \\
 & \mid & \texttt{output}(e)
\end{array}
$$

Here x ranges over identifiers, f over function names, c over confidence-target literals in $[0, 1)$, and m over $\{\texttt{analytical}, \texttt{empirical}(N), \texttt{bayesian}(N)\}$. The symbols $\oplus$ and $\otimes$ stand in for the arithmetic and comparison operators, whose rewrite rules are the standard ones.

8 Operational Semantics

This section gives a small-step operational semantics for YAPPL. The semantics is described in terms of *small-step rules*: each rule takes one expression and rewrites it into a slightly simpler one (the "next state" of the program), and a program is run by repeatedly applying these rules until nothing further can be rewritten. The presentation here is unusual in that an expression never rewrites to a single sampled value; it rewrites to a *distribution*. A deterministic expression rewrites to a *point distribution* concentrated on its value, and a probabilistic construct rewrites to a non-trivial distribution. Only the explicit `sample` method crosses from the distribution world to the stochastic world, consuming one unit of randomness and projecting a distribution to a sample. This presentation directly mirrors the "distributions as first-class values" pillar of §4 and is what makes analytical reasoning about YAPPL programs tractable.

8.1 Configurations

A configuration is a triple $\langle s \,;\, \sigma \,;\, \omega \rangle$ where s is the statement currently executing, σ is a mapping from identifiers to values (the store), and ω is an output buffer. For expressions we use the two-element configuration $\langle e \,;\, \sigma \rangle$. The small-step relation on expressions is written $\langle e \,;\, \sigma \rangle \longrightarrow \langle e' \,;\, \sigma \rangle$, and the statement relation is $\langle s \,;\, \sigma \,;\, \omega \rangle \longrightarrow \langle s' \,;\, \sigma' \,;\, \omega' \rangle$.

We write δ_v for the point distribution at a value v, and $\mathrm{supp}(d)$ for the support of a distribution d. For discrete distributions, $d(v)$ denotes the probability mass at v.

8.2 Core Rewrite Rules

Deterministic expressions rewrite in the standard way (a literal or a variable already evaluated needs no further work). Literals and variable lookups are in normal form:

$$\frac{}{\langle n\,;\,\sigma\rangle \text{ normal}} \qquad \frac{\sigma(x) = v}{\langle x\,;\,\sigma\rangle \longrightarrow \langle v\,;\,\sigma\rangle}\,(\text{E-Var})$$

Arithmetic, comparison, and logical operators reduce left-to-right in the obvious way. The rules are omitted for brevity; the only point worth stating is that every deterministic value is implicitly the point distribution δ_v when that interpretation is forced by a context requiring a `Dist`.

8.3 Distribution Constructors

Distribution constructors reduce their arguments and produce a distribution value directly:

$$\frac{\langle a\,;\,\sigma\rangle \longrightarrow^* \langle n_1\,;\,\sigma\rangle \qquad \langle b\,;\,\sigma\rangle \longrightarrow^* \langle n_2\,;\,\sigma\rangle}{\langle \texttt{uniform}(a,b)\,;\,\sigma\rangle \longrightarrow \langle \texttt{Uniform}(n_1,n_2)\,;\,\sigma\rangle}\,(\text{E-Uniform})$$

$$\frac{\langle p\,;\,\sigma\rangle \longrightarrow^* \langle f\,;\,\sigma\rangle}{\langle \texttt{Bernoulli}(p)\,;\,\sigma\rangle \longrightarrow \langle \texttt{Bernoulli}(f)\,;\,\sigma\rangle}\,(\text{E-Bern})$$

And similarly for `Binomial`, `Geometric`, `Discrete`, and `Beta`. Note carefully that the reduct is a *value* (a distribution literal), not a further expression: there is no hidden sampling here.

8.4 Distribution Operations

Sum. Convolution of two distributions is analytic:

$$\frac{\begin{array}{c}\langle d_1\,;\,\sigma\rangle \longrightarrow^* \langle D_1\,;\,\sigma\rangle \\ \langle d_2\,;\,\sigma\rangle \longrightarrow^* \langle D_2\,;\,\sigma\rangle\end{array}}{\langle d_1 + d_2\,;\,\sigma\rangle \longrightarrow \langle D_1 \oplus D_2\,;\,\sigma\rangle}\,(\text{E-DSum})$$

where $D_1 \oplus D_2$ denotes the convolution: $(D_1 \oplus D_2)(k) = \sum_{i+j=k} D_1(i) \cdot D_2(j)$.

Queries. `expect` and `mean` reduce a fully-formed distribution to a scalar by consulting the analytical formulas. For a discrete distribution D:

$$\frac{\begin{array}{c}\langle d\,;\,\sigma\rangle \longrightarrow^* \langle D\,;\,\sigma\rangle \\ \langle k\,;\,\sigma\rangle \longrightarrow^* \langle v\,;\,\sigma\rangle\end{array}}{\langle d.\texttt{expect}(k)\,;\,\sigma\rangle \longrightarrow \langle D(v)\,;\,\sigma\rangle}\,(\text{E-Expect})$$

$$\frac{\langle d\,;\,\sigma\rangle \longrightarrow^* \langle D\,;\,\sigma\rangle}{\langle d.\texttt{mean}()\,;\,\sigma\rangle \longrightarrow \langle \mathbb{E}[D]\,;\,\sigma\rangle}\,(\text{E-Mean})$$

Sample (the only stochastic rule). `sample` is the unique rule that is non-deterministic at the level of the semantics. Given a fully-formed distribution D, `d.sample()` reduces to an outcome v with probability $D(v)$:

$$\frac{\begin{array}{c}\langle d\,;\,\sigma\rangle \longrightarrow^* \langle D\,;\,\sigma\rangle \\ v \sim D\end{array}}{\langle d.\texttt{sample}()\,;\,\sigma\rangle \longrightarrow \langle v\,;\,\sigma\rangle}\,(\text{E-Sample})$$

All the other probabilistic constructs in the language can be understood as delaying E-Sample for as long as possible or avoiding it entirely.

8.5 Markov Primitives

bind. Given a distribution D and a transition function f, `bind(d, f)` reduces to the analytical marginal of the composite experiment. The rule is:

$$\frac{\langle d\,;\,\sigma\rangle \longrightarrow^{*} \langle D\,;\,\sigma\rangle \qquad \forall s \in \mathrm{supp}(D).\ \langle f(s)\,;\,\sigma\rangle \longrightarrow^{*} \langle F_s\,;\,\sigma\rangle}{\langle \mathtt{bind}(d, f)\,;\,\sigma\rangle \longrightarrow \langle \textstyle\sum_s D(s)\cdot F_s\,;\,\sigma\rangle}\,(\text{E-Bind})$$

No sampling is performed; the resulting distribution is computed by summing the point-multiplications $D(s)F_s$ over the (finite) support of D.

step. `step` is n-fold `bind`:

$$\frac{n = 0}{\langle \mathtt{step}(s_0, f, n)\,;\,\sigma\rangle \longrightarrow \langle \delta_{s_0}\,;\,\sigma\rangle}\,(\text{E-Step-Zero})$$

$$\frac{n > 0}{\langle \mathtt{step}(s_0, f, n)\,;\,\sigma\rangle \longrightarrow \langle \mathtt{bind}(\mathtt{step}(s_0, f, n-1), f)\,;\,\sigma\rangle}\,(\text{E-Step-Succ})$$

8.6 Probabilistic Function Calls

The key rule. A `with confidence` call reduces by dispatching on the error class of the callee, picking a repetition strategy, and iterating round bodies. We use the notation $k_\kappa(c)$ for the round-count function of error class κ and target c (as defined in §4.5).

(E-PbCall-RP). For $\kappa = \mathtt{RP}$, the semantics runs rounds until a `Certain(v)` is produced or $k = k_{\mathtt{RP}}(c)$ rounds have completed. Let $\mathit{round}(f, \overline{v})$ denote a single evaluation of f's body under argument bindings $\overline{v}$:

$$\frac{\overline{v} = \overline{e}\!\downarrow \qquad \mathit{round}(f, \overline{v}) \rightsquigarrow \mathtt{Certain}(u)}{\langle \mathtt{let}\ r, i = f(\overline{e})\ \mathtt{with\ conf} \geq c\,;\,\sigma\,;\,\omega\rangle \longrightarrow \langle \epsilon\,;\,\sigma[r \mapsto u, i \mapsto \mathtt{Info}(1, 1.0)]\,;\,\omega\rangle}$$

$$\frac{\overline{v} = \overline{e}\!\downarrow \qquad \forall j \leq k.\ \mathit{round}(f, \overline{v}) \rightsquigarrow \mathtt{Uncertain}(u_j) \qquad k = k_{\mathtt{RP}}(c)}{\langle \mathtt{let}\ r, i = f(\overline{e})\ \mathtt{with\ conf} \geq c\,;\,\sigma\,;\,\omega\rangle \longrightarrow \langle \epsilon\,;\,\sigma[r \mapsto u_k, i \mapsto \mathtt{Info}(k, 1-2^{-k})]\,;\,\omega\rangle}$$

Here $\overline{e}\!\downarrow$ denotes full evaluation of the argument list to values.

(E-PbCall-coRP). Identical to E-PbCall-RP with the roles of "certain short- circuit" and "uncertain accumulation" adapted for the opposite direction; the formula for k and the computed confidence are unchanged.

(E-PbCall-BPP). For $\kappa = \mathtt{BPP}$, the semantics runs all $k = k_{\mathtt{BPP}}(c) = \lceil -8\ln(1-c)\rceil$ rounds unconditionally, collects the inner boolean of each round (stripping the `Certain` / `Uncertain` wrapper), and takes the majority:

$$\frac{\overline{v} = \overline{e}\!\downarrow \qquad k = k_{\mathtt{BPP}}(c) \qquad b_1, \ldots, b_k = \mathit{unwrap}(\mathit{round}(f, \overline{v}))^k \qquad u = \mathrm{majority}(b_1, \ldots, b_k)}{\langle \mathtt{let}\ r, i = f(\overline{e})\ \mathtt{with\ conf} \geq c\,;\,\sigma\,;\,\omega\rangle \longrightarrow \langle \epsilon\,;\,\sigma[r \mapsto u, i \mapsto \mathtt{Info}(k, 1-e^{-k/8})]\,;\,\omega\rangle}$$

8.7 Confidence Blocks (Bonferroni)

The reserved block form `with confidence >= c { s1; ...; sm; }` where the s_i contain exactly m pb-call statements with bare `with confidence` reduces by rewriting each inner call to use a per-call target of $c' = 1 - (1 - c)/m$:

$$\frac{\begin{array}{c} m = \#\{\text{pb-calls in } \overline{s}\} \\ c' = 1 - (1 - c)/m \\ \overline{s}' = \overline{s}\{\texttt{with conf} \mapsto \texttt{with conf} \geq c'\} \end{array}}{\langle \texttt{with conf} \geq c\ \{\overline{s}\}\,;\,\sigma\,;\,\omega\rangle \longrightarrow \langle \{\overline{s}'\}\,;\,\sigma\,;\,\omega\rangle}\,(\text{E-ConfBlock})$$

The rewrite is purely syntactic: after this single rewrite step, the semantics is the conjunction of the individual E-PbCall rules. The union bound guarantees that the probability any rewritten call returns an incorrect answer is at most $\sum_{i=1}^{m}(1 - c') = m(1 - c)/m = 1 - c$.

The `map` form with confidence distributes its budget the same way, with m equal to the length of the array argument:

$$\frac{\begin{array}{c} \langle a\,;\,\sigma\rangle \longrightarrow^{*} \langle [v_1, \ldots, v_m]\,;\,\sigma\rangle \\ c' = 1 - (1 - c)/m \end{array}}{\langle \texttt{let}\ v = \texttt{map}(f, a)\ \texttt{with conf} \geq c\rangle \longrightarrow \langle \texttt{let}\ v = [f(v_1)@c', \ldots, f(v_m)@c']\rangle}\,(\text{E-MapConf})$$

where $f(v_i)@c'$ denotes a pb-call with per-element target c'.

8.8 `distribution_of`

The analytical mode consults the callee's error class and produces the implied distribution without sampling:

$$\frac{\begin{array}{c} \Gamma(f) = \_ \to^{\kappa} \tau \\ D_\kappa = \text{implied distribution of } \kappa \end{array}}{\langle \texttt{distribution\_of}(f(\overline{e}), \texttt{analytical})\,;\,\sigma\rangle \longrightarrow \langle D_\kappa\,;\,\sigma\rangle}\,(\text{E-DistOf-A})$$

where $D_{\texttt{RP}} = \texttt{Geometric}(0.5)$ and $D_{\texttt{BPP}} = \texttt{Bernoulli}(0.75)$.

The empirical mode runs N single rounds and returns a Bernoulli over the empirical certain-rate:

$$\frac{\begin{array}{c} \overline{v} = \overline{e}\downarrow \\ \text{let } c = |\{j : \mathit{round}(f, \overline{v})_j = \texttt{Certain}(\cdot)\}| \text{ over } N \text{ rounds} \end{array}}{\langle \texttt{distribution\_of}(f(\overline{e}), \texttt{empirical}, N)\,;\,\sigma\rangle \longrightarrow \langle \texttt{Bernoulli}(c/N)\,;\,\sigma\rangle}\,(\text{E-DistOf-E})$$

The Bayesian mode runs N single rounds, tallies certain-vs-uncertain, and returns a Beta posterior with the usual $+1$ Laplace prior:

$$\frac{\begin{array}{c} \overline{v} = \overline{e}\downarrow \\ \text{let } (c, u) = (\#\texttt{Certain}, \#\texttt{Uncertain}) \text{ over } N \text{ rounds} \end{array}}{\langle \texttt{distribution\_of}(f(\overline{e}), \texttt{bayesian}, N)\,;\,\sigma\rangle \longrightarrow \langle \texttt{Beta}(1 + c, 1 + u)\,;\,\sigma\rangle}\,(\text{E-DistOf-B})$$

8.9 Program Execution

A program is executed by reducing its statement sequence in order, with an initial empty store and empty output buffer. Function and enum definitions are first collected into σ in a hoisting pass before any statement rewrite steps begin. Top-level `output` appends its argument (after full evaluation) to the output buffer.

This concludes the operational semantics. Every probabilistic construct in the language has been given a small-step rule that reduces it either analytically (when the answer can be computed from distribution structure) or by iterating a single-round evaluator (when it cannot). The stochastic content of a program is concentrated in two places: the rounds inside a `with confidence` call, and the `sample` method. Everything else is pure.[68]

9 Standard Library Reference

This section is a reference catalogue of every built-in distribution, distribution operation, probabilistic primitive, and numeric built-in function in the language. The types use the conventions of §5. Entries marked "reserved" are defined in the specification but not yet available in the prototype.

9.1 Distribution Constructors

Name	Signature	Description
`uniform`	$\texttt{int}, \texttt{int} \to \texttt{Dist}\langle\texttt{int}\rangle$	Discrete uniform on $\{a, a{+}1, \ldots, b\}$.
`uniformContinuous`	$\texttt{float}, \texttt{float} \to \texttt{Dist}\langle\texttt{float}\rangle$	Continuous uniform on $[a, b)$.
`Discrete`	$\overline{\tau{:}\texttt{float}} \to \texttt{Dist}\langle\tau\rangle$	Explicit PMF over keys of any type τ supporting equality. Keys may be enum variants.
`Bernoulli`	$\texttt{float} \to \texttt{Dist}\langle\texttt{bool}\rangle$	$\Pr[\texttt{true}] = p$.
`Binomial`	$\texttt{int}, \texttt{float} \to \texttt{Dist}\langle\texttt{int}\rangle$	Number of successes in n Bernoulli(p) trials.
`Geometric`	$\texttt{float} \to \texttt{Dist}\langle\texttt{int}\rangle$	Number of trials until first success, per-trial probability p.
`Beta`	$\texttt{float}, \texttt{float} \to \texttt{Dist}\langle\texttt{float}\rangle$	Conjugate prior/posterior on $[0, 1]$. Produced by the `bayesian` mode of `distribution_of`.
`Point`	$\tau \to \texttt{Dist}\langle\tau\rangle$	Degenerate distribution at a single outcome. *Reserved*; the same effect is obtained with `Discrete(x: 1.0)`.

Table 5: Built-in distribution constructors.

9.2 Distribution Combinators

Name	Signature	Description
`+` (convolution)	$\texttt{Dist}\langle\tau\rangle, \texttt{Dist}\langle\tau\rangle \to \texttt{Dist}\langle\tau\rangle$	Distribution of the sum of independent samples, for $\tau \in \{\texttt{int}, \texttt{float}\}$. Analytical.
`bind`	$\texttt{Dist}\langle\tau\rangle, (\tau \to \texttt{Dist}\langle\tau\rangle) \to \texttt{Dist}\langle\tau\rangle$	Monadic bind for distributions, threaded with a transition function. Analytical.
`step`	$\tau, (\tau \to \texttt{Dist}\langle\tau\rangle), \texttt{int} \to \texttt{Dist}\langle\tau\rangle$	n-fold `bind` from a delta at the initial state.
`product`	$\texttt{Dist}\langle\tau_1\rangle, \texttt{Dist}\langle\tau_2\rangle \to \texttt{Dist}\langle(\tau_1, \tau_2)\rangle$	Independent Cartesian product. *Reserved.*
`d.map(f)`	$\texttt{Dist}\langle\tau\rangle, (\tau \to \tau') \to \texttt{Dist}\langle\tau'\rangle$	Pushforward along a pure function. *Reserved* as a method; do not confuse with the statement-level `map` form.

Table 6: Distribution combinators.

9.3 Distribution Queries and Observations

Method	Signature	Description
`d:sample()`	`Dist`$\langle\tau\rangle \to \tau$	The only primitive in the language that does something probabilistic: draws one random sample from d.
`d:expect(k)`	`Dist`$\langle\tau\rangle, \tau \to$ `float`	$\Pr[X = k]$, reported as an exact rational where possible. Discrete distributions only.
`d:mean()`	`Dist`$\langle\tau\rangle \to$ `float`	Analytical expected value.
`d:min()`	`Dist`$\langle\tau\rangle \to \tau$	Lower bound of the support. Uniform distributions only.
`d:max()`	`Dist`$\langle\tau\rangle \to \tau$	Upper bound of the support. Uniform distributions only.
`d:entropy()`	`Dist`$\langle\tau\rangle \to$ `float`	Shannon entropy in nats. *Reserved.*
`d:visualise()`	`Dist`$\langle\tau\rangle \to$ `Visualisation`	Renders the distribution: ASCII histogram on the CLI, SVG in the web playground.

Table 7: Distribution queries and observations.

9.4 Equality

Distribution equality exists in YAPPL because the natural questions one asks about a probabilistic computation almost always involve a comparison: "does the empirical distribution of my pb function match the analytical one?", "is the stationary distribution of this chain the one I expected?", "does this Bayesian posterior agree with my prior plus the observed data?". Without first-class equality on distributions, every such question would have to be reduced by hand to a comparison of individual probabilities, which is exactly the kind of bookkeeping the language is designed to remove. The exact form (`==`) covers cases where the two distributions should agree by construction; the approximate form (`~=`) covers cases where small numerical or sampling differences are acceptable.

Operator	**Signature**	**Description**
`==`	$\texttt{Dist}\langle\tau\rangle, \texttt{Dist}\langle\tau\rangle \to \texttt{bool}$	Structural (exact) equality.
`~=`	$\texttt{Dist}\langle\tau\rangle, \texttt{Dist}\langle\tau\rangle \to \texttt{bool}$	Approximate equality: TV distance for discrete distributions, moment comparison for continuous. Default tolerance 0.05.
`~= ... within t`	$\texttt{Dist}\langle\tau\rangle, \texttt{Dist}\langle\tau\rangle, \texttt{float} \to \texttt{bool}$	Same as `~=` with a programmer-specified tolerance t.

Table 8: Distribution equality operators.

9.5 Probabilistic Function Primitives

Form	Shape	Description
pb-call	`let r, i = f(...) with confidence >= c`	Runs f until its answer is correct with confidence at least c, picking a repetition strategy from f's error class. Binds the result and the `Info` record.
union-bound map	`let v = map(f, arr) with confidence >= c`	Element-wise pb-call over an array with Bonferroni-corrected per-element target.
confidence block	`with confidence >= c { ... }`	Reserved: distributes c across m inner bare pb-calls via the union bound.
`distribution_of`, applied	`distribution_of(f(...), mode)`	Reifies the per-round distribution of f into a `Dist` value. Three modes: `analytical`, `empirical`, `bayesian`.
`distribution_of`, curried	`distribution_of(f, mode)`	Reserved: produces a function from f's argument types to `Dist<return-type-of-f>`.
`Certain(v)`	round return	Definitive answer. May only appear as the returned expression of a pb function body.
`Uncertain(v)`	round return	Tentative answer. May only appear as the returned expression of a pb function body.

Table 9: Probabilistic function primitives.

9.6 Markov Chain Primitives

The Markov chain primitives exist to support the expressiveness of YAPPL on a larger class of probabilistic computations than a single `pb function` call: namely, computations whose state evolves stochastically over discrete time. They were added so that small Markov chains (such as the weather example in §6.3) could be written in YAPPL without forcing the programmer to encode the transition matrix by hand or to drop down to a host language for matrix multiplication. The two primitives below are deliberately minimal; richer combinators (continuous-time chains, hidden states, observation operators) are reserved for future revisions.

Form	Signature	Description
`bind`	$\texttt{Dist}\langle\tau\rangle, (\tau \to \texttt{Dist}\langle\tau\rangle) \to \texttt{Dist}\langle\tau\rangle$	One-step monadic bind for distributions. Analytical.
`step`	$\tau, (\tau \to \texttt{Dist}\langle\tau\rangle), \texttt{int} \to \texttt{Dist}\langle\tau\rangle$	n-fold bind from a delta at the initial state.

Table 10: Markov chain primitives.

9.7 Numeric Built-ins

The numeric built-ins exist to support the expressiveness of YAPPL on the small set of randomised algorithms that the language was designed to demonstrate. In particular, they were added so that a faithful proof-of-concept implementation of the Solovay–Strassen primality test (§6.1) could be written entirely in YAPPL without the programmer having to drop down to a host language for the bits of number theory the test depends on. The current set is therefore deliberately narrow: a future revision could replace it with a more general standard-library mechanism, but the present shape covers the algorithms shipped with the language.

Name	Signature	Description
`jacobi`	`int`, `int` → `int`	Jacobi symbol $(a\|n)$. Returns -1, 0, or 1. Used in Solovay–Strassen (§6.1).
`mod_exp`	`int`, `int`, `int` → `int`	Modular exponentiation: $\texttt{base}^{\texttt{exp}} \bmod m$.

Table 11: Numeric/symbolic built-in functions.

9.8 Info Records

A `pb function` call returns an `Info` record alongside its result:

```
Info { rounds: int, confidence: float }
```

`rounds` is the number of rounds the runtime actually executed (which is at most the round count implied by the target); it can be strictly less than that when the chosen repetition strategy short-circuits. `confidence` is the actual confidence achieved, which is at least the requested target.

The current shape of the `Info` record is intentionally narrow: two scalar fields are enough to describe what every existing repetition strategy in the language does. If probabilistic functions were expanded in a future revision (for example, to allow correlated rounds, mixed error classes, or per-round witness traces), the natural place for the extra information to live would be in additional `Info` fields, and the existing two fields would be retained as the lowest common denominator that every repetition strategy can fill in.

9.9 Output Forms

Form	Shape	Description
`output`	`output(e)`	Appends a rendered view of e to the program's output buffer. Only available at the top level.

Table 12: Top-level output forms.

This completes the YAPPL language specification.

B Original Project Specification

The original Project Specification submitted at the start of the year is reproduced on the following pages. It states the original problem, objectives, technical and planning methodology, timetable, risks, and ethical considerations against which the delivered project is evaluated.

CS351 Project Specification:
A Probabilistic Programming Language

Edward Denton
5515605

16th October 2025

1 Abstract

An implementation of a probabilistic programming language with a focus on implementing and evaluating programming language features for probabilistic programming.

2 Problem Statement

Problems that can be efficiently solved by a probabilistic Turing machine belong to the complexity class RP, and there are problems in RP which are not known to be efficiently solvable using non-probabilistic Turing machines. This project aims to develop a programming language with probability as a core feature, to allow easy construction of algorithms to solve problems in RP. This would involve exploration of programmatically modelling probability distributions, programming language features to ameliorate the experience of designing randomised algorithms, and building an interpreter/compiler for the language. By evaluating and implementing different programming language features, this project aims to contribute to the field of probabilistic programming and its applications in predictive models, intrusion detection, and machine learning.

3 Objectives

- Produce a specification for the language (how is it structured? what features?)
- Build an interpreter for the language
- Evaluate language features (such as precomputation, macros etc) in terms of performance and how it lends itself to making probabilistic programming easier (**An issue here is that how good a language feature is may be quite subjective**)

4 Technical Methodology

An interpreter is chosen to develop the language in order to cut down on the scope of the project and make it more feasible and easier to focus on the other features of the project.
The downside of this is that the interpreter will not be as performant as its hypothetical compiler.

Rust is chosen as the language to write the programming language in.

Generally, Rust is chosen because of its performance and safety, which enables it to be an ideal language for a large program.

Rust also has Nom which is a parser combinator library, which seems ideal for our purposes.

Git and GitHub are used to back-up code (see Resources and Risks).

5 Planning Methodology

A largely agile methodology will be used due to the high chance of aims changing and therefore deadlines needing to be amended, with a rough initial plan (see Timetable) to guide the progress of the project (and ensure deadlines are adhered to).

To enable the flexibility required for this, as outlined below, tasks will have a small buffer zone (usually a weekend) before deadlines and also between consecutive tasks.

6 Timetable

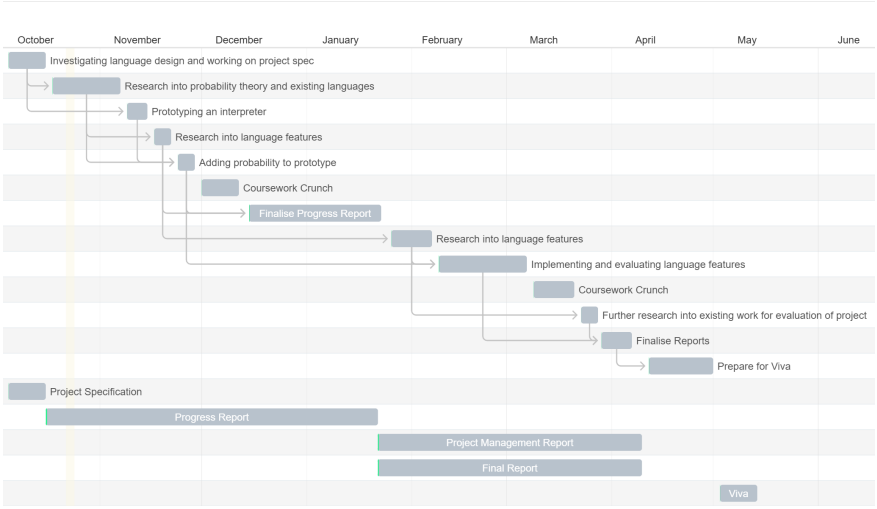

Figure 1: Gantt Chart to show timeline for the project

7 Resources and Risks

A modern computing device is required to develop the language, particularly one that is capable of running Rust.

Risk: Laptop breaks down.
Mitigation: Can switch to an alternative device - any of the DCS machines (since they also have Rust installed, so it would be quick to switch). Use Git & GitHub to back-up my code, so that no (or minimal) progress is lost.

Risk: Project code is lost.
Mitigation: Mentioned above - use Git & GitHub to back-up code.

A device with internet access will be needed for research.

8 Ethical Considerations

No direct ethical considerations.

Indirect considerations are on what the language is used for - for instance, the language could be used for finance and defense. I would argue that this is the same for pretty much any programming language.

No ethical consent needed. If a survey is conducted (for assessing whether the , ethical consent will be needed.