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.

(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);
}
```

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.

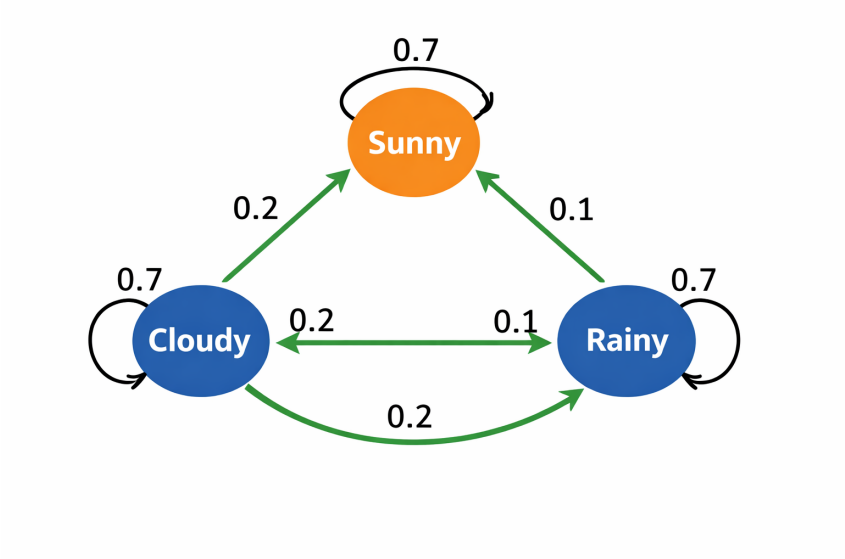

Figure 1: The weather Markov chain as a state-transition diagram. Each arrow is annotated with the per-step probability of moving from one weather state to another; `step` iterates this transition function and converges the initial delta distribution toward a fixed point (the stationary distribution).

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:

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

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:

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

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)`.

```
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 Naïve The `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).

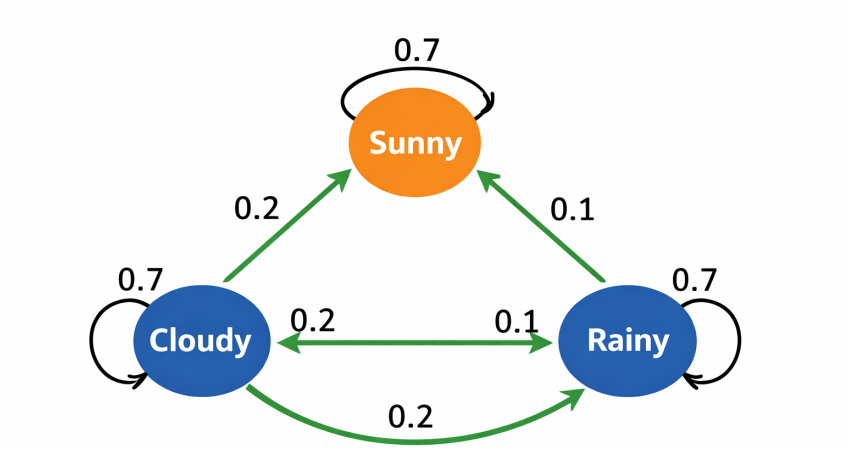

Figure 2: The same weather chain as a state-transition diagram, repeated here for the worked example.

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)?
```

```
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.

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.