1 Введение

ProbabilityScript — это специализированный язык сценариев, предназначенный для упрощения описания и выполнения вероятностных экспериментов и статистических вычислений. Он предоставляет минималистичный синтаксис для моделирования случайных процессов, сбора выборочных данных и расчёта статистических характеристик. Данный язык и его интерпретатор разработаны с целью облегчить исследование концепций теории вероятностей и математической статистики, позволяя пользователю фокусироваться на постановке задачи, а не на низкоуровневых деталях реализации.

Основная сфера применения ProbabilityScript — учебные и исследовательские задачи, требующие проведения множества повторяющихся симуляций, вычисления таких величин, как средние значения, дисперсии, медианы и т. д., а также анализа полученных результатов. Цель реализации данного языка заключалась в создании простого в использовании и расширяемого инструмента, который можно применять для проверки законов больших чисел, моделирования случайных величин, оценки характеристик распределений и других вероятностных экспериментов.

Интерпретатор ProbabilityScript написан на C++ и выполняет пошаговую интерпретацию сценариев, написанных на этом языке. При разработке акцент сделан на ясности архитектуры и корректности реализации основных алгоритмов, таких как генерация случайных чисел и вычисление статистик. Несмотря на ориентированность на конкретную область применения, язык спроектирован с возможностью дальнейшего расширения, например, добавления новых типов данных или конструкций управления.

2 Синтаксис и грамматика

2.1 Обзор конструкций языка

Программа на ProbabilityScript состоит из последовательности операторов, выполняемых последовательно. Каждый оператор может быть либо простым (например, присваивание или вызов функции), либо составным (например, условная конструкция или цикл). Для группировки нескольких операторов в блок используется фигурные скобки {}, что позволяет объединять их в единое целое там, где синтаксис требует один оператор (например, тело цикла или условного оператора). В языке используются английские ключевые слова и идентификаторы. Для разделения операторов применяется либо точка с запятой ‘;‘, либо перевод строки (аналогично многим скриптовым языкам), что обеспечивает удобочитаемость кода.

Поддерживаются следующие виды операторов:

- **Присваивание**: сохранение значения выражения в переменную.
- **Условный оператор** `if`: выполнение блока операторов при выполнении некоторого условия.
- **Цикл повторения** `repeat`: многократное выполнение блока операторов заданное число раз.
- **Оператор вывода** `print`: печать значения выражения на экран.

- **Оператор сбора данных** `collect`: сохранение значения выражения в специальную выборку для последующей статистической обработки.
- **Оператор вывода статистик** `print_stat`: вычисление и вывод основных статистических параметров накопленной выборки.

Кроме вышеперечисленных, язык поддерживает вызов встроенных функций (например, генераторов случайных чисел), которые могут использоваться в выражениях наравне с переменными и литеральными значениями. Каждая функция имеет фиксированную сигнатуру (список ожидаемых аргументов) и возвращает определённое значение (в случае генераторов — случайное число с заданными параметрами распределения).

2.2 Выражения и операторы

Выражения в ProbabilityScript задают правила вычисления числовых значений. Поддерживаются стандартные арифметические операции: сложение ‘+‘, вычитание ‘-‘, умножение ‘*‘ и деление ‘/‘. Также допускается унарный минус для смены знака. Приоритет операторов стандартный: сначала выполняются умножение и деление, затем сложение и вычитание (см. таблицу **??**). Можно использовать круглые скобки для явного задания порядка вычислений.

Кроме арифметических операций, выражения могут содержать операции сравнения: равно ‘==‘, не равно ‘!=‘, меньше ‘<‘, больше ‘>‘, меньше или равно ‘<=‘, больше или равно ‘>=‘. Результатом сравнения также является вещественное число: 1.0 (истина) или 0.0 (ложь). Это позволяет напрямую использовать результаты сравнений в дальнейших вычислениях или условных выражениях. Отдельного логического типа в языке не вводится — логическое значение представляется числом (как в языке Си), причём любое ненулевое числовое значение трактуется как *истина*, а нуль как *ложь*. Логические операции (`&&`, `||`, `!`) специально не реализованы; при необходимости логическое И и ИЛИ могут быть выражены через арифметические операции (например, $a\&\&b$ эквивалентно $(a * b) \neq 0$).

Числовые литералы записываются в десятичном формате (например, `10`, `3.14`). Идентификаторы (имена переменных и функций) должны начинаться с буквы и могут содержать буквы, цифры и символ подчёркивания, например `x`, `sum_1`. Идентификаторы чувствительны к регистру (т. е. `X` и `x` — разные имена). Следующие слова зарезервированы и не могут использоваться в качестве имён переменных: `if`, `else`, `repeat`, `print`, `collect`, `print_stat`, `uniform`, `normal`, `seed` и др.

Таблица 1: Приоритет и ассоциативность операторов ProbabilityScript

Категория	Операторы	Ассоциативность
Скобки	`( )`	– (явно задают порядок)
Унарные	`-` (унарный)	справа налево
Множение/деление	`*`, `/`	слева направо
Сложение/вычитание	`+`, `-`	слева направо
Сравнения	`>`, `<`, `>=`, `<=`	не ассоциативны
Равенство	`==`, `!=`	не ассоциативны

2.3 Синтаксис операторов

Ниже приведена формальная грамматика языка ProbabilityScript в нотации, близкой к EBNF:

Программа	::=	*СписокОператоров*
СписокОператоров	::=	*Оператор* { (“;” | \n) *Оператор* }
Оператор	::=	*Присваивание* | *If* | *Repeat* | *Print* | *Collect* | *PrintStat*
Присваивание	::=	*Идентификатор* “=” *Выражение*
If	::=	“if” “(” *Выражение* “)” *Блок*
Repeat	::=	“repeat” *Выражение Блок*
Print	::=	“print” *Выражение*
Collect	::=	“collect” *Выражение*
PrintStat	::=	“print_stat”
Блок	::=	“{” *СписокОператоров* “}” | *Оператор*
Выражение	::=	*Выражение OpEq Выражение*
	|	*Выражение OpRel Выражение*
	|	*Выражение OpAdd Выражение*
	|	*Выражение OpMul Выражение*
	|	*OpUnary Выражение*
	|	“(” *Выражение* “)”
	|	*Число* | *Идентификатор* | *ВызовФункции*
OpEq	::=	“==” | “!=”
OpRel	::=	“<” | “>” | “<=” | “>=”
OpAdd	::=	“+” | “-”
OpMul	::=	“*” | “/”
OpUnary	::=	“-”
ВызовФункции	::=	*Идентификатор* “(” [*Аргументы*] “)”
Аргументы	::=	*Выражение* { “,” *Выражение* }

Как видно из грамматики, присваивание имеет форму `<идентификатор> = <выражение>`. Оператор `if` требует в скобках условное выражение и исполняет последующий блок, если значение выражения истинно (не равно нулю). Блок после `if` может быть либо одним оператором без фигурных скобок, либо набором операторов в { }. Конструкции `else` в текущей версии языка не предусмотрено: при необходимости её можно имитировать, используя несколько `if` с противоположными условиями или добавлять расширение в будущем (см. раздел **??**).

Цикл `repeat` выполняет указанный блок операторов определённое число раз. После ключевого слова `repeat` ожидается выражение, вычисляющее число итераций. Это выражение округляется до целого (отбрасыванием дробной части) в случае вещественного значения. Если результат меньше или равен 0, цикл просто не выполняется ни разу. Пример: `repeat 100 { collect X }` выполнит вложенный блок 100 раз.

Оператор `print` вычисляет значение выражения и выводит его (например, на консоль или стандартный поток вывода). Оператор `collect` вычисляет выражение и добавляет полученное значение в специальную коллекцию (выборку) внутри среды выполнения для последующего статистического анализа. Наконец, оператор `print_stat` инициирует расчёт основных статистических показателей по накопленным данным и их вывод. Как правило, `print_stat` вызывается после серии операций `collect`, и выводит такие величины, как объём выборки, среднее значение,

дисперсия, среднеквадратическое отклонение и медиана (подробнее в разделе **??**). `print_stat` не требует указания параметров — он всегда относится ко всей текущей накопленной выборке.

2.4 Встроенные функции

ProbabilityScript включает ряд встроенных функций, расширяющих возможности языка. Их можно вызывать внутри выражений аналогично математическим функциям. Ниже перечислены основные из них с формальной сигнатурой и семантикой:

- `seed(`*`S`*`)` — инициализирует генератор случайных чисел заданным целочисленным *seed* (тип аргумента автоматически приводится к целому). Вызывать эту функцию имеет смысл один раз в начале сценария, если требуется воспроизводимость результатов. Возвращаемого значения не имеет (может считаться равным 0).

- `uniform(a, b)` — возвращает случайное вещественное число, равномерно распределённое в промежутке $[a, b)$. Аргументы a и b задают границы интервала (вещественные). Если $a \geq b$, функция может возвращать всегда a либо генерировать ошибку (пользователю рекомендуется задавать $a < b$ для корректной работы).

- `normal(`*`mean, stddev`*`)` — возвращает случайное вещественное число, распределённое по нормальному закону с математическим ожиданием *mean* и среднеквадратическим отклонением *stddev*. При *stddev* ≤ 0 генерируется ошибка (дисперсия должна быть положительной).

Перечисленные функции позволяют легко включать в сценарии генерацию случайных данных. Например, выражение `x + uniform(0, 1)` возвращает значение переменной `x`, увеличенное на случайную величину в диапазоне $[0, 1)$.

3 Типы данных

В текущей версии ProbabilityScript имеется единственный тип данных для хранимых и вычисляемых величин — **вещественное число двойной точности** (тип `double` в C++). Каждая переменная содержит 64-битное значение с плавающей запятой:contentReference[oaicite:0]index=0, что обеспечивает порядка 15 десятичных значащих цифр точности и диапазон значений примерно от 10^{-308} до 10^{308}:contentReference[oaicite:1]inc
Данный формат соответствует стандарту IEEE 754 (Double Precision) и широко используется в вычислительной технике.

Все числовые литералы в исходном коде интерпретируются как значения типа double. Если требуется представлять целочисленные данные, они также хранятся в формате с плавающей запятой (т. е. 5 будет представлено как 5.0). Арифметические операции выполняются в соответствии с правилами IEEE 754, включая возможное возникновение специальных значений: бесконечностей ($+\infty$, $-\infty$) при переполнении или делении на ноль и NaN (Not a Number) при недопустимых операциях (например, 0/0). Интерпретатор не заменяет поведение стандартной арифметики C++: такие случаи не вызывают исключений, и пользователь должен самостоятельно следить за корректностью получаемых результатов.

Внутренняя репрезентация вещественных чисел — формат с порядком и мантиссой, реализуемый аппаратно. Это означает, что операции над числами осуществляются быстро, однако имеются ограничения на точность (например, не все десятичные дроби представимы точно в двоичном формате). Модель вычислений соответствует принятой в языке C++: используется округление к ближайшему representable значению, возможно накопление погрешности при последовательных операциях. Впрочем, для большинства задач моделирования случайных процессов такой точности более чем достаточно.

Отсутствие отдельных типов для целых, логических или строковых данных упрощает язык и реализацию интерпретатора, но накладывает ограничения: например, невозможно напрямую работать с текстовыми данными или структурами. Тем не менее, единый числовой тип делает язык однородным и предсказуемым: все выражения возвращают число, и все переменные хранят числа. В заключительном разделе документации (раздел **??**) рассмотрены возможности расширения системы типов в будущем.

4 Лексический анализ

На этапе лексического анализа входной исходный текст программы преобразуется в последовательность токенов (элементарных лексем), удобных для последующего синтаксического анализа. Лексический анализатор (сканер) проходит по исходному коду символ за символом и распознаёт токены согласно заданным правилам.

Основные виды токенов в ProbabilityScript:

- **Идентификаторы**: последовательности букв, цифр и символа "_" (подчёркивания), не начинающиеся с цифры. Примеры: `x`, `X2`, `sample_mean`. Используются для обозначения переменных и имён функций.

- **Ключевые слова**: зарезервированные идентификаторы, обозначающие конструкции языка: `if`, `else`, `repeat`, `print`, `collect`, `print_stat`, `uniform`, `normal`, `seed` и т. д. Они распознаются так же, как идентификаторы, но на этапе лексического анализа отмечаются особым типом токена и не требуют создания переменной.

- **Числовые литералы**: десятичные числа, возможно содержащие точку (для дробной части). Пример: `12`, `3.14`, `0.5`. Экспоненциальная нотация (например, `1e-3`) может не поддерживаться (в текущей реализации упор сделан на простоту). Литерал обрабатывается как значение типа double.

- **Операторы и разделители**: символы, имеющие специальное значение в синтаксисе, включая `+`, `-`, `*`, `/`, `==`, `!=`, `>`, `<`, `>=`, `<=`, `=`, `(`, `)`, `{`, `}`, `,`, `;`.

- **Прочие токены**: пробельные символы (пробел, табуляция, перевод строки) и комментарии игнорируются сканером. В данной версии языка комментарии как таковые не определены, поэтому любой текст после символа # или // не распознаётся специально и приведёт к лексической ошибке, если не является частью синтаксиса.

Лексер реализован по принципу *детерминированного конечного автомата* (ДКА). При чтении исходного текста он находится в некотором состоянии (например, начальном, в котором не накоплено ничего) и в зависимости от текущего символа либо переходит в новое состояние, либо завершает распознавание текущего токена. Например, упрощённо:

- При встрече буквы в начальном состоянии сканер переходит в состояние "чтение идентификатора" и будет считывать последующие буквы/цифры.
- При встрече цифры в начальном состоянии сканер переходит в состояние "чтение числа" и считывает последовательность цифр, при встрече точки — остаётся в том же состоянии (для дробной части).
- Пробельные символы в начальном состоянии игнорируются (пропускаются) и не приводят к созданию токена.
- Специальные символы (например, `+`, `-`) мгновенно распознаются как отдельные токены (при этом `-` требует проверки, не является ли он частью числа или унарным оператором — обычно в лексере `-` всегда токен оператора, а унарный минус различается потом на этапе синтаксического анализа).
- Составные операторы из двух символов (например, `==`, `<=`) распознаются путем проверки сразу после первого символа: лексер заглядывает на следующий символ и при совпадении объединяет их в один токен.

Алгоритм работы лексера можно описать следующим псевдокодом:

```
while (не конец входа) {
    ch = next_char();
    if (ch является пробелом или управляющим символом новой строки) {
        continue; // пропустить
    } else if (буква(ch)) {
        lexeme = ch;
        while (буквaИлиЦифра(peek_char())) {
            lexeme += next_char();
        }
        if (lexeme является ключевым словом) {
            emit_token(TOKEN_KEYWORD, lexeme);
        } else {
            emit_token(TOKEN_IDENTIFIER, lexeme);
        }
    } else if (цифра(ch)) {
        lexeme = ch;
        has_dot = false;
        while (цифра(peek_char()) || (peek_char() == '.' && !has_dot)) {
            next = next_char();
            if (next == '.') has_dot = true;
            lexeme += next;
        }
        emit_token(TOKEN_NUMBER, lexeme);
    } else {
```

```
        switch (ch) {
            case '+': emit_token(TOKEN_PLUS, "+"); break;
            case '-':
                if (цифра(peek_char())) {
                    // особый случай: литерал с ведущим минусом как отдельный токен '
                    emit_token(TOKEN_MINUS, "-");
                    continue;
                } else {
                    emit_token(TOKEN_MINUS, "-"); break;
                }
            case '=':
                if (peek_char() == '=') { next_char(); emit_token(TOKEN_EQ, "=="); }
                else emit_token(TOKEN_ASSIGN, "=");
                break;
            case '!':
                if (peek_char() == '=') { next_char(); emit_token(TOKEN_NEQ, "!="); }
                else throw LexError("неожиданный символ '!'");
                break;
            // ... другие односимвольные токены
            default:
                throw LexError("неизвестный символ");
        }
    }
}
```

В приведённом фрагменте `emit_token(TYPE, text)` означает создание токена определённого типа с указанным текстовым значением, `peek_char()` — просмотр следующего символа без потребления. Реальная реализация может отличаться, но суть остаётся той же: мы идентифицируем классы символов и на основе этого принимаем решения о формировании токенов.

Если лексер встречает последовательность символов, не подпадающих ни под одно правило (например, `$` или кириллические буквы вне строкового литерала), он генерирует лексическую ошибку. Сообщение об ошибке обычно содержит номер строки/позиции и описание проблемы (“неизвестный символ”). На лексическом уровне также выявляются слишком большие числовые литералы (при превышении диапазона double, однако библиотека C++ сама преобразует их в `inf`) и другие простые ошибки.

5 Синтаксический анализ

Синтаксический анализатор (парсер) отвечает за разбор последовательности токенов согласно грамматике языка и построение абстрактного синтаксического дерева (AST). В данном проекте реализован **рекурсивный спуск** — простой и наглядный метод парсинга, подходящий для относительно простой грамматики ProbabilityScript. Каждое правило грамматики соответствует функции парсера: например, функция `parseExpression()` разбирает нетерминал *Выражение*, а `parseStatement()` — *Оператор*. Парсер последовательно вызывает эти функции по мере необходимости, обходя древовидно структуру грамматики.

Ниже, для иллюстрации, приведена структура основных узлов AST и соответствующих им конструкций языка:

- **Expr** — абстрактный класс (интерфейс) для всех узлов, представляющих выражения. Имеет потомков:
  - **NumberExpr** (литеральное число) — хранит значение типа double.
  - **VariableExpr** (идентификатор переменной) — хранит имя переменной.
  - **BinaryExpr** (бинарная операция) — содержит указатели на левое и правое подвыражения и тип операции (например, `+`, `<`, `==`).
  - **UnaryExpr** (унарная операция) — содержит подвыражение и тип (например, унарный минус).
  - **CallExpr** (вызов функции) — хранит имя вызываемой функции и список узлов-аргументов (выражений).
- **Stmt** — абстрактный класс для узлов-операторов (statements). Основные подклассы:
  - **AssignStmt** (присваивание) — хранит имя переменной и выражение-значение.
  - **IfStmt** (условный оператор) — содержит выражение-условие и узел-блок (или одиночный оператор), исполняемый при истинности условия.
  - **RepeatStmt** (цикл) — содержит выражение-счётчик итераций и узел-блок, который нужно повторно выполнять.
  - **PrintStmt** (вывод) — содержит выражение, значение которого требуется вывести.
  - **CollectStmt** (сбор данных) — содержит выражение, значение которого нужно добавить в выборку.
  - **PrintStatStmt** (вывод статистики) — не хранит дополнительных данных (оперирует глобальной выборкой среды выполнения).
  - **BlockStmt** (блок кода) — содержит список дочерних узлов-операторов, представляющих операторы внутри фигурных скобок.

Парсер строит AST, последовательно создавая объекты вышеперечисленных классов. Например, при разборе присваивания `x = uniform(0, 1)` будет создан узел AssignStmt с именем “x” и дочерним узлом-выражением CallExpr для вызова `uniform` с двумя узлами NumberExpr в качестве аргументов.

Для управления порядком операций парсер учитывает приоритеты. Реализация рекурсивного спуска обычно разделяет разбор выражений по уровням приоритета: например, `parseTerm()` для считывания факторов умножения/деления, `parseFactor()` для базовых элементов (число, скобки, вызов функции). При разборе бинарных операций применяется принцип левоассоциативности: после разбора левого операнда и оператора парсер рекурсивно разбирает правый операнд и при необходимости продолжает чтение последующих операций того же уровня.

Если на входе обнаруживается последовательность, не соответствующая грамматике, парсер генерирует **синтаксическую ошибку**. Например, отсутствие закрывающей скобки или ключевого слова там, где оно ожидается, вызовет выброс исключения с сообщением об ошибке (обычно указываются текущий неразобранный

токен и ожидания парсера). После возникновения синтаксической ошибки интерпретация сценария прерывается.

Построение AST является критически важной фазой, поскольку именно на основе дерева будет осуществляться выполнение программы на следующем этапе. AST устраняет неоднозначности исходного текста (например, жёстко фиксирует порядок вычислений в соответствии с приоритетами) и предоставляет удобную структуру данных для интерпретатора.

6 Выполнение программы (интерпретация)

Интерпретатор ProbabilityScript выполняет код, обходя построенное синтаксическое дерево и производя вычисления согласно семантике языка. Модель исполнения основана на **обходе AST** (вычислительном дереве): для каждого узла вычисляются (или выполняются) его подузлы, после чего применяется операция, соответствующая данному узлу.

6.1 Вычисление выражений

Выражения вычисляются в соответствии с ожидаемой семантикой математических операций:

- **Литерал (NumberExpr)**: непосредственное значение возвращается как результат.
- **Переменная (VariableExpr)**: интерпретатор обращается к среде выполнения, чтобы получить текущее значение переменной с данным именем. Если переменная не была ранее присвоена, генерируется ошибка времени выполнения (использование неинициализированной переменной).
- **Бинарная операция (BinaryExpr)**: сначала вычисляются левое и правое подвыражения (рекурсивный обход поддеревьев), затем к полученным значениям применяется операция. Арифметические операции (+, -, *, /) возвращают числовой результат. Операции сравнения (>, <, >=, <=, ==, !=) возвращают 1.0, если условие истинно, или 0.0, если ложно. Деление на ноль на уровне интерпретатора не перехватывается специально — при попытке вычислить $x/0$ будет получено значение `inf` или `NaN` в соответствии с IEEE 754.
- **Унарная операция (UnaryExpr)**: вычисляет подвыражение и, например, меняет знак (для унарного минуса) у полученного значения.
- **Вызов функции (CallExpr)**: интерпретатор вычисляет все выражения-аргументы (слева направо), затем выполняет предопределенную функцию с этими аргументами и возвращает полученный результат. Например, `uniform(0,1)` при вызове генерирует случайное число и возвращает его.

При вычислении выражений стоит отметить модель истинности: **любое ненулевое число трактуется как истина**. Это проявляется, например, при вычислении условного выражения `if` — результат выражения проверяется на неравенство нулю. В то же время, выражения сравнения сами возвращают 0 или 1, так что вложенные сравнения работают интуитивно ожидаемо.

6.2 Выполнение операторов

Интерпретатор обрабатывает операторы верхнего уровня программы последовательно, обновляя при необходимости состояние (содержимое переменных, накопленные данные выборки и пр.):

- **Присваивание (AssignStmt)**: вычисляется выражение значения, затем полученное число сохраняется в среде выполнения под заданным именем переменной. Если такой переменной ещё не было, она создаётся; иначе её старое значение заменяется новым.

- **Условный оператор (IfStmt)**: вычисляется выражение-условие; если результат отличен от 0 (истина), интерпретатор выполняет вложенный оператор или блок. Если результат равен 0 (ложь), вложенный блок пропускается. (Напомним, ветка `else` не реализована в текущей версии, поэтому при ложном условии просто ничего не происходит.)

- **Цикл (RepeatStmt)**: вычисляется выражение-счётчик. Из него извлекается целочисленное значение (например, приводится через `int` путём отбрасывания дробной части). Пусть результат равен N. Если $N \leq 0$, цикл не выполняется. Если $N > 0$, интерпретатор повторно (в цикле `for` или аналогично) выполняет вложенный блок N раз. Каждая итерация цикла может изменять переменные, вызывать функции, собирать данные и т. д. как обычный последовательный код.

- **Вывод (PrintStmt)**: вычисляется вложенное выражение; затем его значение выводится пользователю. Конкретная реализация может выводить число в стандартный поток вывода (консоль) с определённой форматированием (например, фиксированного или экспоненциального вида). После вывода, выполнение продолжается.

- **Сбор данных (CollectStmt)**: вычисляет выражение и добавляет его результат в текущую выборку (массив) внутри среды выполнения. Этот оператор не производит видимого эффекта сразу, но влияет на последующий результат `print_stat`. Обычно `collect` вызывается внутри цикла, чтобы накопить серию наблюдений случайной величины.

- **Вывод статистики (PrintStatStmt)**: при выполнении этого оператора интерпретатор обращается к среде выполнения для вычисления статистических параметров по накопленной выборке. Рассчитываются, как минимум, объём выборки (количество собранных значений), выборочное среднее, выборочная дисперсия, среднеквадратическое отклонение и медиана. Затем эти показатели выводятся на экран в читаемом формате, после чего (по реализации) выборка может быть очищена (либо сохраняться для последующих вычислений, если не очищать). Поведение реализации: как правило, после `print_stat` выборка сбрасывается, чтобы следующая серия `collect` начала новый эксперимент.

В процессе интерпретации состояние программы хранится и изменяется в структуре **среды выполнения** (о ней в следующем разделе). Интерпретатор не выполняет никаких оптимизаций кода во время исполнения (таких как пропуск повторных вычислений) — каждый оператор обрабатывается явно. Это упрощает поведение и

улучшает предсказуемость (в ущерб скорости при очень больших объёмах данных, что, однако, не критично для учебных задач).

Следует отметить, что **ложные** (нулевые) условия и пустые циклы просто приводят к тому, что вложенные операторы пропускаются, без ошибок. Однако некоторые ошибки (например, деление на ноль, выход за диапазон) неявно обработаются как особые числовые значения, о чём было сказано выше.

7 Среда выполнения

Среда выполнения (*runtime environment*) представляет собой совокупность структур данных, хранящих состояние интерпретируемой программы во время её выполнения. В данной реализации среда включает:

- Таблицу переменных и их значений.
- Структуру для хранения собранных значений (выборки) при вызове `collect`.
- Генератор случайных чисел (и его параметры, например, текущее зерно генерации).

7.1 Хранение переменных

Переменные сохраняются в динамической структуре (например, `std::unordered_map<string, double>` или аналогичном словаре), сопоставляющей именам (ключам) числовые значения. При выполнении оператора присваивания, если имя отсутствует в таблице, создаётся новая запись; если присутствует, её значение обновляется. Все переменные глобальные (в текущей версии нет вложенных областей видимости или локальных переменных функций). Область видимости переменных — весь сценарий после их первого определения.

Доступ к переменной (`VariableExpr`) осуществляется путём поиска имени в таблице. Это операция $O(\log n)$ или $O(1)$ в среднем (в зависимости от реализации контейнера). Если имя не найдено, интерпретатор генерирует ошибку (с информацией о некорректном имени переменной). Переменные не типизированы явно, хранят всегда вещественные числа. Память под них выделяется по мере добавления записей; освобождение (в текущей реализации) происходит только по завершении программы (или при явном удалении, что не предусмотрено на уровне языка).

7.2 Механизм `collect` и накопление выборки

При каждом выполнении оператора `collect` вычисленное значение добавляется в специальную коллекцию, представляющую собой список (или динамический массив) чисел. В реализации используется, например, `std::vector<double>` или собственная структура последовательности. Изначально, до начала сбора данных, коллекция пуста. Каждый вызов `collect` делает `collection.push_back(value)`. Таким образом накапливается выборка значений, которую можно рассматривать как результаты повторных наблюдений некоторой случайной величины.

Оператор `print_stat` обращается к этой коллекции для вычисления статистик. После вывода статистики, по замыслу, выборка может быть очищена (например, `collection.clear()`) для начала новой серии сбора, хотя это зависит от реализации.

В нашем интерпретаторе после `print_stat` коллекция действительно очищается, чтобы избежать повторного включения прошлых данных в новые расчёты.

Предусмотрена возможность собирать большие объёмы данных; стандартные контейнеры способны динамически расширяться при добавлении элементов. Ограничением является лишь объём памяти.

7.3 Генерация случайных чисел

Для реализации функций `uniform` и `normal` используется генератор псевдослучайных чисел. В среде выполнения хранится текущее **seed** (зерно генерации) и генератор (например, `std::mt19937` или вызовы `rand()` из <cstdlib>). Функция `seed(s)` задаёт начальное значение генератора: в случае использования `rand()` вызывается `srand(s)`, а при использовании <random> - инициализируется объект генератора заданным seed.

Для равномерного распределения `uniform(a, b)` берётся либо стандартная функция `rand()`, нормированная на $[0, 1)$ (с последующим масштабированием $a + (b - a) * U(0, 1)$), либо генератор `std::uniform_real_distribution` из библиотеки <random>. Результат приводится к double и возвращается.

Для нормального распределения `normal(mean, stddev)` возможны разные методы. В данной реализации использован метод Бокса–Мюллера: генерируются две независимые равномерные величины U_1, U_2 на $[0, 1)$, затем вычисляется $Z = \sqrt{-2 \ln U_1} \cos(2\pi U_2)$, имеющее стандартное нормальное распределение. Результат масштабируется: $X = mean + Z \cdot stddev$ и возвращается пользователю. При каждом вызове `normal` происходит вычисление нового независимого случайного значения.

Важно отметить, что при фиксированном `seed` последовательность возвращаемых псевдослучайных чисел детерминирована (что полезно для воспроизводимости результатов), а при разном `seed` или если `seed` не вызывать вовсе (часто генератор инициализируют текущим временем), последовательности будут различаться.

Точность и распределение генератора соответствуют используемым библиотечным функциям. Например, `rand()` обычно генерирует 31-битные числа, что может не давать идеальной статистической равномерности, но для учебных целей приемлемо. При необходимости генератор можно заменить на более качественный без изменения логики интерпретатора.

8 Статистический модуль

Одной из ключевых возможностей ProbabilityScript является автоматическое вычисление статистических характеристик по собранным данным. После выполнения серии операторов `collect` можно вызвать `print_stat`, чтобы получить сводку по выборке. Реализованы следующие статистики:

- **count** (объём выборки N): общее количество значений, добавленных в выборку на момент вызова `print_stat`.
- **mean** (выборочное среднее $\bar{x}$): $\bar{x} = \dfrac{1}{N} \sum_{i=1}^{N} x_i$. Вычисляется суммированием всех значений и делением на N.

- **var** (выборочная дисперсия s^2): мера разброса значений, вычисляемая по формуле $s^2 = \frac{1}{N}\sum_{i=1}^{N}(x_i - \bar{x})^2$. В реализации для эффективности может использоваться эквивалентная формула $s^2 = \frac{1}{N}\sum x_i^2 - \bar{x}^2$, но она менее численно устойчива. Для повышения устойчивости при больших N можно применять алгоритм на основе первичных и вторичных моментов или метод Вельфорда.

- **stddev** (среднеквадратическое отклонение σ): квадратный корень из дисперсии, $\sigma = \sqrt{s^2}$. Показывает среднее отклонение значений от среднего.

- **median** (медиана): величина, отделяющая верхнюю половину выборки от нижней. Вычисляется следующим образом: выборка сортируется, и если N нечётно, медиана равна среднему элементу, если N чётно, медиана берётся как полусумма двух средних значений:contentReference[oaicite:2]index=2. Формально: при чётном N median $= \frac{x_{(N/2)} + x_{(N/2+1)}}{2}$, при нечётном — $x_{((N+1)/2)}$ (где $x_{(k)}$ — k-й элемент упорядоченного по возрастанию набора):contentReference[oaicite:3]index=3.

Алгоритмы вычисления этих характеристик следующие:

- **Объём выборки** подсчитывается при добавлении (можно хранить счётчик и просто выводить его значение).

- **Сумма** $\sum x_i$ и **сумма квадратов** $\sum x_i^2$ могут накапливаться на лету при каждом `collect`. Тогда среднее и дисперсия могут быть вычислены в `print_stat` в $O(1)$. Наш интерпретатор, однако, для простоты вычисляет обе суммы в момент вывода, проходя один раз по массиву значений (сложность $O(N)$).

- **Среднее** $\bar{x}$ получается делением суммы на N.

- **Дисперсия** s^2 вычисляется по определению или альтернативной формуле. Важным моментом является численная стабильность: прямой подсчёт $\frac{\sum x_i^2}{N} - \bar{x}^2$ может привести к потере значимых цифр при вычитании близких величин. Для уменьшения ошибки предпочтительно использовать двупроходный алгоритм (сначала находить $\bar{x}$, затем суммировать квадраты отклонений) или алгоритм Вельфорда. В этом проекте, учитывая умеренные объёмы данных, был реализован двупроходный подход: после получения $\bar{x}$ выполняется второй цикл для вычисления $\sum(x_i - \bar{x})^2$.

- **Медиана** вычисляется сортировкой копии выборки и выбором центрального элемента (или двух) как описано выше. Время работы сортировки $O(N \log N)$, что для целей данного языка (как правило, N не слишком велико) вполне допустимо.

Численная устойчивость вычисления средней и дисперсии обеспечивается выбором алгоритма. Для большинства практических случаев точности double достаточно, однако при N в сотни тысяч может проявляться накопление ошибки. Использование улучшенных алгоритмов (например, метод Кэхэна для суммирования) могло бы улучшить точность суммирования, но в текущей реализации это не критично.

Результаты статистических вычислений выводятся обычно с определённым форматированием (например, фиксированное число знаков после запятой). Пример вывода:

```
Count = 1000
Mean = 0.5021
Variance = 0.0813
Std.Dev = 0.2851
Median = 0.5006
```

В данном примере показаны названия статистик и их значения. Такие результаты позволяют пользователю быстро оценить характеристики распределения случайной величины, данные для которой он собрал с помощью `collect`.

9 Обработка ошибок

Интерпретатор ProbabilityScript различает несколько типов ошибок:

- **Лексические ошибки**: обнаруживаются на этапе лексического анализа, когда входной текст содержит недопустимые символы или последовательности. Например, встреча неизвестного символа (не буквы, не цифры и не поддерживаемого знака пунктуации) вызовет ошибку. Также сюда можно отнести чрезмерно длинные идентификаторы (если бы накладывались ограничения на длину) или некорректный формат числового литерала (например, `..5`).

- **Синтаксические ошибки**: возникают на этапе разбора, когда последовательность токенов не может быть применена к грамматике языка. Примеры: отсутствует закрывающая скобка `)`, лишняя фигурная скобка, неожиданное ключевое слово, неправильный порядок токенов (например, `= x 5` вместо `x = 5`). При синтаксической ошибке парсер обычно генерирует исключение `ParseException` с сообщением, указывающим текущий токен и ожидаемые варианты.

- **Ошибки времени выполнения**: выявляются в процессе исполнения программы. Сюда относятся:

  – Обращение к неопределённой переменной (отсутствует в таблице символов).

  – Деление на ноль в явном виде (хотя, как упоминалось, на уровне double это не приводит к исключению, но может рассматриваться как логическая ошибка).

  – Недопустимые аргументы функций: например, `normal(0, -5)` (отрицательное стандартное отклонение), `uniform(5, 1)` (левая граница больше правой — в текущей реализации это не вызывает исключения, но приводит к некорректному смыслу).

  – Вызов `print_stat` при пустой выборке. В такой ситуации интерпретатор выдаёт сообщение об ошибке (недостаточно данных для статистики) и останавливает выполнение.

- **Логические ошибки**: это ошибки в логике программы пользователя, которые интерпретатор не может автоматически обнаружить. Например, неправильно заданная формула, лишний или пропущенный вызов `collect`, неверная интерпретация результатов. Эти ошибки не генерируют исключений, но приводят к неправильным результатам. Ответственность за их устранение лежит на пользователе.

При обнаружении ошибки интерпретатор формирует диагностическое сообщение. В сообщении указывается тип ошибки (лексическая, синтаксическая или выполнения) и описание проблемы. Для синтаксических и лексических ошибок обычно сообщается номер строки и позиция в строке, где обнаружена ошибка, чтобы упростить отладку сценария. Например: `Syntax error: unexpected token '}' at line 5, col 3`.

После выдачи сообщения выполнение программы прекращается. Интерпретатор не пытается продолжать исполнение после ошибки, так как состояние программы может быть некорректным. Исключение составляют лишь некоторые нефатальные ситуации, трактуемые как предупреждения (в текущей версии таких нет, но гипотетически можно игнорировать повторную инициализацию seed и т. п.).

Стоит отметить, что большинство ошибок выявляется на ранних этапах (лексика и синтаксис). Если исходный код успешно проходит парсинг, то далее остаётся небольшой набор возможных ошибок выполнения, перечисленных выше. Логические же ошибки находятся вне контроля интерпретатора, и отладка таких ошибок сводится к анализу получаемых результатов и сопоставлению их с ожидаемыми.

10 Архитектура реализации

Весь проект интерпретатора ProbabilityScript состоит из нескольких модулей, отражающих основные этапы обработки кода:

- **Лексер** (`lexer.cpp`) — содержит функции и структуры для чтения исходного текста и генерации токенов. Определяет типы токенов (например, перечисление `TokenType` с вариантами: IDENTIFIER, NUMBER, KEYWORD_IF, OPERATOR_PLUS, и т. д.) и структуру токена (тип + значение + позиция). Лексер обеспечивает метод `getNextToken()`, возвращающий следующий токен при каждом вызове.

- **Парсер** (`parser.cpp`) — реализует функции рекурсивного спуска, описанные в предыдущем разделе. Он использует лексер для получения токенов (либо через явные вызовы `getNextToken()`, либо на основе текущего "глядывания" `currentToken`). В случае обнаружения синтаксической ошибки парсер возбуждает исключение `ParseException`.

- **AST** (`ast.h`, `ast.cpp`) — содержит определения классов узлов синтаксического дерева, таких как Expr и Stmt, и их наследников. Каждый класс имеет поля для хранения необходимых данных (например, у BinaryExpr — `op`, `left`, `right`), а также может определять виртуальный метод `execute()` или `evaluate()` для выполнения (в некоторых реализациях интерпретатора код выполнения можно включить прямо внутрь узлов AST).

- **Интерпретатор** (`interpreter.cpp`) — основной модуль, который либо рекурсивно обходит AST, либо содержит реализацию visitor-паттерна для применения операций. В нашем случае, например, реализована функция `executeStatement(Stmt* stmt)`, которая по типу оператора вызывает соответствующую обработку, и `evaluateExpression(Expr* expr)` для вычисления выражений. Интерпретатор управляет средой выполнения: читает и записывает переменные, управляет коллекцией собранных значений, вызывает генератор случайных чисел.

- **Среда выполнения** (`environment.h`) — включает структуры для хранения переменных и выборки, а также функции-утилиты: например, `addSample(double x)` (добавить значение в выборку), `clearSamples()` (очистить выборку), `getMean()` (вычислить среднее), `getVariance()`, `getMedian()` и т. д. Этот модуль инкапсулирует логику статистических вычислений, позволяя интерпретатору просто вызывать `env.printStats()` для вывода результатов.

- **Модуль матриц** (`Matrix_MAS.h`, `Matrix.cpp`) — реализует класс `Matrix<T>` для математических операций над матрицами, включая алгоритмы решения СЛАУ, обратной матрицы и подсчёта нормы и числа обусловленности. Этот модуль не напрямую используется интерпретатором для исполнения скриптов (так как язык пока не поддерживает переменные-матрицы), однако он был разработан в рамках проекта и может быть интегрирован при расширении языка (см. §??). Например, можно представить, что в будущей версии появится тип `matrix` и функции для работы с ним, тогда реализованные алгоритмы будут кстати.

Структура кода разбивает ответственность по классическим фронтам компилятора/интерпретатора: лексический анализ, синтаксический анализ, выполнение. Такое разделение облегчает отладку и развитие проекта. Например, можно модифицировать или улучшать алгоритм лексера, не затрагивая парсер или интерпретатор. Добавление новых встроенных функций чаще всего требует правок только в интерпретаторе (добавить обработку нового имени в `CallExpr` или как отдельный случай в eval). Добавление новых синтаксических конструкций скажется на парсере и потребует новых классов AST, но ядро интерпретации остаётся прежним.

Язык C++ выбран для реализации благодаря эффективности и богатой стандартной библиотеке (включая `<random>` для генерации чисел, `<algorithm>` для сортировки при вычислении медианы и т. п.). Классы и функции разделены по файлам, как указано выше, для удобства сопровождения.

Наконец, модуль **тестирования** (`Tests.h`, `main.cpp`) содержит набор функций, проверяющих корректность основных компонентов: динамических массивов, списков, матричных алгоритмов и др. В функции `main` проекта вызываются `TestDynamicArray()`, `TestMatrixIdentity()` и подобные, чтобы убедиться в правильности низкоуровневых частей. После успешного прохождения тестов интерпретатор может быть использован для запуска пользовательских скриптов (в рамках данного задания, возможно, интерактивный ввод или чтение файла не реализованы, но могли бы быть добавлены).

11 Алгоритмы

В ходе реализации в модуле класса матриц были разработаны и применены несколько важных алгоритмов линейной алгебры:

11.1 Решение СЛАУ (метод Гаусса с выбором главного элемента)

Для решения системы линейных алгебраических уравнений $A\mathbf{x} = \mathbf{b}$ (где A — заданная квадратная матрица размерности n, $\mathbf{b}$ — вектор правых частей), используется

модификация метода Гаусса — метод Гаусса с частичным выбором главного элемента. Алгоритм состоит из двух этапов: прямого хода (элиминации) и обратного хода (нахождения решения). В нашей реализации фактически выполняется алгоритм Гаусса–Жордана: на этапе прямого хода матрица приводится к единичной, одновременно преобразуется вектор $\mathbf{b}$ к виду решения.

Основные шаги:

1. Для каждого столбца i (от 1 до n) на этапе прямого хода выбирается строка $p \geq i$ такая, что $|A_{p,i}|$ максимален среди $|A_{k,i}|$ для $k = i, \ldots, n$. Если найденный главный элемент равен 0 (с точностью до порога, например 10^{-12}), система считается вырожденной (матрица A сингулярна) и алгоритм прерывается с ошибкой. Иначе строка p меняется местами со строкой i (это и есть **частичный выбор по столбцу** — максимизация элемента в текущем столбце):contentReference[oaicite:4]index=4.

2. Строка i (после обмена) нормируется: все её элементы (включая правую часть) делятся на главный элемент $A_{i,i}$, так что на диагонали получается 1.

3. Из всех остальных строк вычитается i-я строка, умноженная на соответствующий коэффициент, чтобы обнулить элемент в i-м столбце во всех строках, кроме i-й. Этот процесс выполняется как для строк ниже, так и выше текущей (особенность алгоритма Гаусса–Жордана), превращая столбец i в столбец единицы (на позиции i) и нулей.

После завершения этого процесса слева от вертикальной черты (если рассматривать расширенную матрицу $[A|b]$) получается единичная матрица, а справа — решение $\mathbf{x}$. Таким образом, задача решается без отдельного этапа обратного хода (он был встроен в обнуление элементов как выше, так и ниже главного элемента).

Сложность данного алгоритма $O(n^3)$ операций в худшем случае. Частичный выбор главного элемента улучшает численную устойчивость метода, избегая ситуации, когда выбранный наивно ведущий элемент очень мал и приводит к большим ошибкам за счёт деления на него:contentReference[oaicite:5]index=5. Следует отметить, что существуют случаи, когда и с выбором по столбцу метод может дать большие погрешности, но в общем эта эвристика существенно повышает надёжность решения.

11.2 Нахождение обратной матрицы

Для вычисления обратной матрицы A^{-1} используется схожий подход: решается n систем $A\mathbf{x}_j = \mathbf{e}_j$, где $\mathbf{e}_j$ — j-й столбец единичной матрицы (базисный вектор с 1 на позиции j и 0 на остальных). Практическая реализация: формируется расширенная матрица $[A|I]$ размерности $n \times 2n$ и к ней применяется метод Гаусса. После приведения A к единичной, правая половина расширенной матрицы станет A^{-1}. Наш код фактически делает именно это, используя уже написанный решатель СЛАУ: функция `getInverse()` создаёт единичную матрицу I того же размера и вызывает `solveSLAE(`I`)`, которая возвращает матрицу X, являющуюся решением $AX = I$, то есть искомой A^{-1}.

Если матрица A вырожденна (определитель 0), то на этапе решения СЛАУ будет сгенерировано исключение, и обращение к `getInverse()` также завершится с

ошибкой. В остальных случаях корректность вычисленной обратной матрицы можно проверить, перемножив $A \cdot A^{-1}$ и сравнив с I (в тестах проекта такая проверка присутствует).

11.3 Матричная норма и обусловленность

В классе `Matrix` реализована функция `getKNorm()`, вычисляющая так называемую 1-норму матрицы (норму столбца): $\|A\|_1 = \max\limits_{1 \le j \le n} \sum\limits_{i=1}^{n} |a_{ij}|$:contentReference[oaicite:6]index=6. То есть для каждого столбца суммируются модули элементов, и берётся наибольшая из этих сумм. Эта норма является одной из часто используемых операторных норм матрицы. Также реализована функция `getCondKNorm()`, вычисляющая число обусловленности матрицы в этой норме. Число обусловленности определяется как $\mathrm{cond}_1(A) = \|A\|_1 \cdot \|A^{-1}\|_1$:contentReference[oaicite:7]index=7. Если матрица необратима, число обусловленности считается бесконечностью.

Число обусловленности даёт оценку чувствительности решения СЛАУ к изменениям коэффициентов матрицы или правой части. Малое cond (порядка 1) означает хорошо обусловленную матрицу, большие значения cond указывают на близость к вырождению и возможность больших ошибок в решении при небольших возмущениях данных. Наш интерпретатор не использует эти величины напрямую в процессе расчётов, однако модуль матриц предоставляет их для анализа, и при расширении языка до поддержки матричных операций эти функции могут оказаться полезными.

В целом, разработанные алгоритмы подтверждены тестированием (например, TestMatrixIdentity проверяет, что метод `getInverse()` корректно находит обратную для матрицы, TestMatrixIdentity также косвенно проверяет решение СЛАУ). Алгоритмы решения СЛАУ и нахождения обратной имеют кубическую сложность и работают эффективно на матрицах малой и средней размерности. Алгоритмы вычисления норм имеют квадратичную сложность ($O(n^2)$ для перебора всех элементов). Все они реализованы на чистом C++ с использованием базовых контейнеров (возможно, обобщённые на тип `T` с помощью шаблонов, что допускает использование `float`, `double` или типа произвольной точности при необходимости).

12 Заключение

В данной документации представлен полный обзор языка сценариев ProbabilityScript, его семантики, а также внутренних деталей реализации интерпретатора. Разработанный язык уже предоставляет базовый набор возможностей для моделирования случайных процессов и сбора статистики, однако существует ряд направлений для расширения и улучшения:

- **Расширение синтаксиса**: Добавление ветви `else` к условному оператору `if` позволило бы обрабатывать альтернативный случай в тех же конструкциях. Также можно ввести циклы типа `while` (с условием, проверяемым перед каждой итерацией) или `for` (с счётчиком), что повысит выразительность языка.

- **Новые типы данных**: Введение целочисленного типа (например, `int`) могло бы повысить эффективность и корректность в тех случаях, когда требуется работать с целыми (например, подсчёт). Отдельный логический тип (`bool`) позволил бы явно выражать условия. В перспективе можно добавить и тип `matrix` для работы с матрицами непосредственно из сценария, используя уже реализованные алгоритмы (обратная матрица, решение СЛАУ и пр.).

- **Больше встроенных функций**: Например, генерация случайных чисел по другим законам распределения (экспоненциальному, биномиальному и т. д.), дополнительные математические функции (тригонометрические, логарифмы). Статистический модуль можно расширить расчетом доверительных интервалов, квантилей, тестов на распределение.

- **Улучшение производительности**: Текущая интерпретация работает в однопоточном режиме. Можно рассмотреть возможность распараллеливания циклов `repeat` при очень больших N (если каждый цикл независим). Кроме того, можно реализовать компиляцию сценария в байт-код или машинный код (JIT-компиляция), что ускорило бы выполнение при многократных запусках одних и тех же расчётов.

- **Интеграция с внешними системами**: Добавление возможностей ввода/вывода из файлов, графического отображения гистограмм собранных данных или взаимодействия с языками типа Python/Matlab для более широкого применения.

- **Обработка ошибок**: Развитие системы сообщений об ошибках, добавление предупреждений (например, предупреждать о вызове `print_stat` на пустой выборке, вместо немедленного прекращения исполнения).

В своём текущем состоянии ProbabilityScript успешно справляется с задачами, для которых он создавался: позволяет с минимальным кодом выполнить серии испытаний, собрать статистику и вывести ключевые характеристики. Он демонстрирует концепцию создания специализированного языка для ограниченной предметной области (Domain-Specific Language) и может служить основой для дальнейших экспериментов студентов и разработчиков.

Опираясь на описанную архитектуру, дальнейшее развитие языка может происходить инкрементально. Например, добавление нового оператора потребует:

1. Изменить грамматику парсера (добавить новый кейс в `parseStatement`).
2. Создать новый класс узла AST (если необходимо).
3. Дополнить функцию интерпретации (например, метод `executeStatement`) для обработки нового узла.
4. (Опционально) Скорректировать лексер, если вводится новое ключевое слово или символ.

Благодаря модульности системы, такие изменения локальны.

Подводя итог, проект ProbabilityScript продемонстрировал полный цикл разработки от формального определения языка до его программной реализации. Техническая документация, представленная выше, даёт целостное представление о принципах работы интерпретатора и может служить руководством для разработчиков, желающих разобраться или усовершенствовать данный язык.