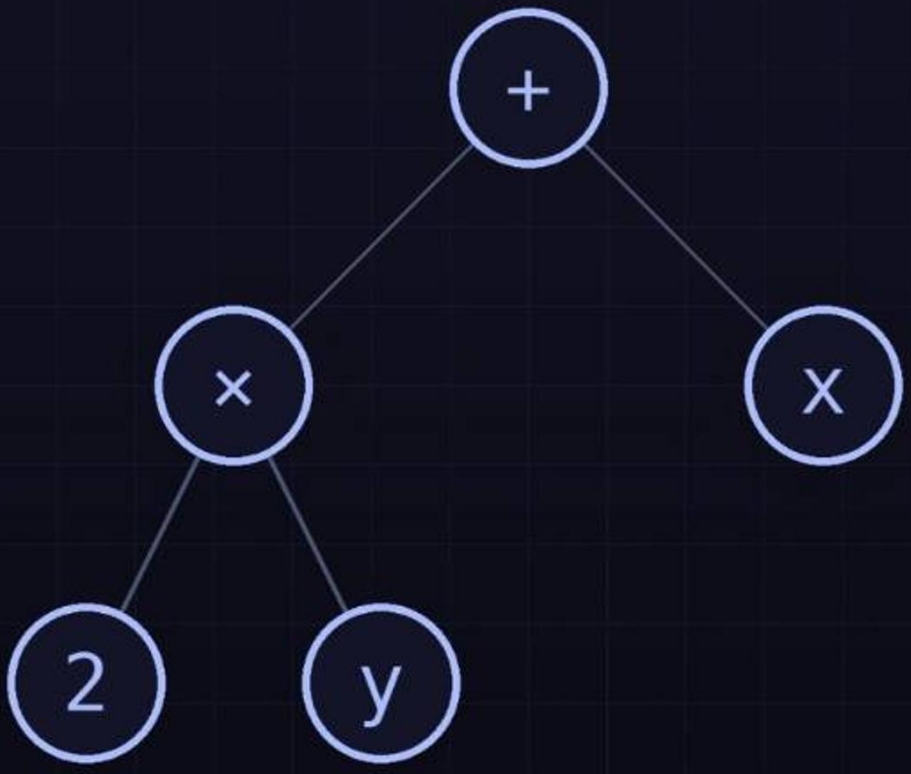
Python C++23 Rust Java



Интерпретатор

Interpreter

Алексей Примак

Алексей Примак

Интерпретатор. Паттерн GoF на Python, C++23, Rust и Java

«Автор»
2026

Примак А.

Интерпретатор. Паттерн GoF на Python, C++23, Rust и Java / А. Примак — «Автор», 2026

Как дать пользователю возможность задавать правила и выражения, не позволяя ему выполнять произвольный код? «Интерпретатор» показывает построение простого языка из грамматики и объектов-выражений. На примерах правил скидок, фильтров и маршрутизации разобраны идеи паттерна GoF. Затем приводятся реализации на Python, C++23, Rust и Java, варианты расширения, практические модификации и самостоятельные работы.

© Примак А., 2026
© Автор, 2026

Содержание

Введение 5
Как читать 6
Глава 15. Реализация на Python 7
15.1. Зачем нужен интерпретатор 7
15.2. Участники и схема 8
15.3. Классический пример: калькулятор с переменными 9
Конец ознакомительного фрагмента. 13

Интерпретатор. Паттерн GoF на Python, C++23, Rust и Java

Введение

Эта книга посвящена одному паттерну проектирования — **Интерпретатор** (Interpreter). Он относится к каталогу «Банды четырёх» (Gang of Four, GoF). Здесь паттерн разобран сразу в четырёх реализациях: на Python, C++23, Rust и Java. Каждая реализация — отдельная часть книги: сначала задача и общая схема, затем код, модификации, гибридные варианты и самостоятельные работы.

Как читать

Основы каждого языка здесь не повторяются: они собраны в книгах серий по соответствующему языку. Разделы каждой части нумеруются заново, поэтому ссылка «раздел 1.2» указывает на раздел той же части, где она сделана.

Глава 15. Реализация на Python

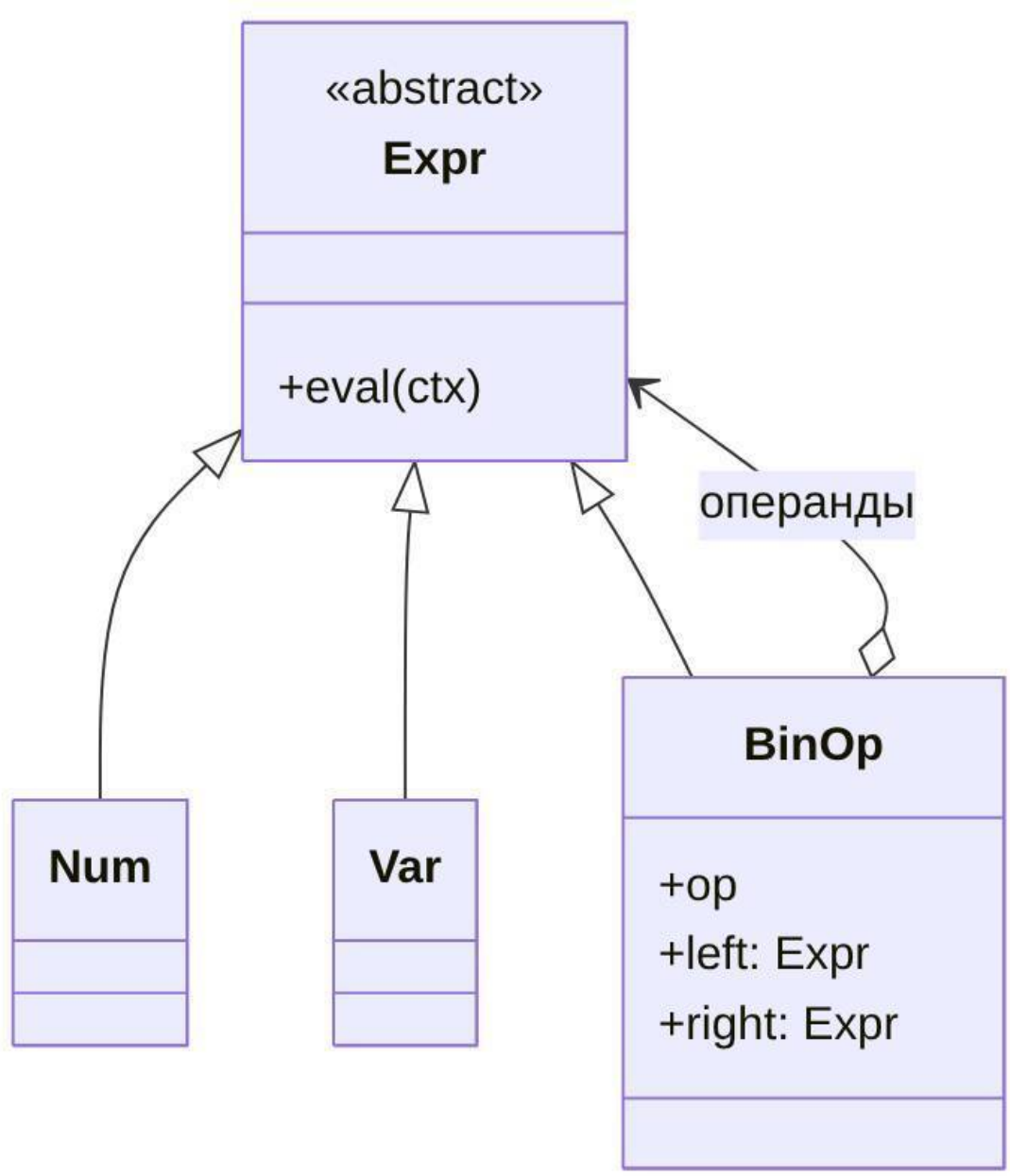
15.1. Зачем нужен интерпретатор

Маркетологи хотят сами задавать правила скидок: «сумма больше 5000 и клиент постоянный или промокод SALE». Аналитики пишут фильтры для отчётов, администраторы — правила маршрутизации писем. Каждый раз переписывать код под новое правило нельзя, а давать людям писать Python — опасно.

Интерпретатор задаёт грамматику небольшого языка и представляет каждое предложение языка деревом объектов. Каждый вид узла знает, как вычислить себя. Конечные узлы (числа, переменные) — терминальные выражения, составные (и, или, +) — нетерминальные. Дерево вычисляется рекурсивно в заданном контексте.

Паттерн хорошо подходит для простых языков: правил, фильтров, формул, команд. Для сложных языков нужны генераторы парсеров и компиляторы, но идея дерева выражений остаётся той же — модуль ast самого Python представляет программы именно так.

15.2. Участники и схема



Abstract Expression— общий интерфейс eval(context).
Terminal Expression— листья грамматики.
Nonterminal Expression— правила, содержащие другие выражения.
Context— глобальная информация для вычисления (значения переменных).
Client— строит дерево (обычно парсером) и запускает вычисление.

15.3. Классический пример: калькулятор с переменными

Файл 01_calculator.py:

01_calculator.py — Python

```python
from __future__ import annotations

import operator
import re
from dataclasses import dataclass


class Expr:
    def eval(self, ctx: dict[str, float]) -> float:
        raise NotImplementedError


@dataclass
class Num(Expr):
    value: float

    def eval(self, ctx):
        return self.value


@dataclass
class Var(Expr):
    name: str

    def eval(self, ctx):
        if self.name not in ctx:
            raise NameError(f"переменная {self.name!r} не задана")
        return ctx[self.name]


@dataclass
class BinOp(Expr):
    op: str
```

Листинг 15.1 *— Классический пример: калькулятор с переменными*

01_calculator.py (продолжение) Python

```
    left: Expr
    right: Expr

    OPS = {"+": operator.add, "-": operator.sub, "*": operator.
        mul, "/": operator.truediv}

    def eval(self, ctx):
        return self.OPS[self.op](self.left.eval(ctx), self.
            right.eval(ctx))


# Грамматика:  expr := term (('+'|'-') term)*
#              term := atom (('*'|'/') atom)*
#              atom := число | имя | '(' expr ')'
class Parser:
    def __init__(self, text: str) -> None:
        self.tokens = re.findall(r"\d+\.?\d*|\w+|[-+*/()]",
            text)
        self.i = 0

    def peek(self) -> str | None:
        return self.tokens[self.i] if self.i < len(self.tokens)
            else None

    def take(self) -> str:
        tok = self.peek()
        if tok is None:
            raise SyntaxError("неожиданный конец выражения")
        self.i += 1
        return tok

    def parse(self) -> Expr:
        e = self.expr()
        if self.peek() is not None:
```

Листинг 15.1 (продолжение) *— Классический пример: калькулятор с переменными*

01_calculator.py (продолжение) Python

```
            raise SyntaxError(f"лишний токен {self.peek()!r}")
        return e

    def expr(self) -> Expr:
        node = self.term()
        while self.peek() in ("+", "-"):
            node = BinOp(self.take(), node, self.term())
        return node

    def term(self) -> Expr:
        node = self.atom()
        while self.peek() in ("*", "/"):
            node = BinOp(self.take(), node, self.atom())
        return node

    def atom(self) -> Expr:
        tok = self.take()
        if tok == "(":
            node = self.expr()
            if self.take() != ")":
                raise SyntaxError("ожидалась ')'")
            return node
        if tok[0].isdigit():
            return Num(float(tok))
        if tok.isidentifier():
            return Var(tok)
        raise SyntaxError(f"неожиданный токен {tok!r}")


if __name__ == "__main__":
    ctx = {"price": 1200, "qty": 3, "discount": 0.1}
    for src in ["price * qty", "price * qty * (1 - discount) +
        300", "2 + 3 * 4", "(2 + 3) * 4", "price * tax", "
        2 + * 3"]:
```

Листинг 15.1 (продолжение) *— Классический пример: калькулятор с переменными*

01_calculator.py (продолжение) Python

```
        try:
            print(f"{src:<36} = {Parser(src).parse().eval(ctx):
                g}")
        except (SyntaxError, NameError) as e:
            print(f"{src:<36} ! {e}")
    print(Parser("a + b * 2").parse())
```

```
$ python 01_calculator.py
price * qty                          = 3600
price * qty * (1 - discount) + 300   = 3540
2 + 3 * 4                            = 14
(2 + 3) * 4                          = 20
price * tax                          ! переменная 'tax' не задана
2 + * 3                              ! неожиданный токен '*'
BinOp(op='+', left=Var(name='a'), right=BinOp(op='*',
    left=Var(name='b'), right=Num(value=2.0)))
```

Листинг 15.1 (продолжение) *— Классический пример: калькулятор с переменными*

Разбор. Работа разделена на две части. Парсер — рекурсивный спуск — превращает текст в дерево: каждому правилу грамматики соответствует метод, и приоритет операций задаётся тем, что term (умножение) вызывается из expr (сложения). Само вычисление — дело узлов: BinOp.eval вычисляет детей и применяет операцию. Одно дерево можно вычислить в разных контекстах, не разбирая текст повторно.

Последняя строка показывает само дерево: умножение b * 2 стало правым поддеревом сложения, потому что связывает сильнее.

Конец ознакомительного фрагмента.

Текст предоставлен ООО «Литрес».

Прочитайте эту книгу целиком, купив полную легальную версию на Литрес.

Безопасно оплатить книгу можно банковской картой Visa, MasterCard, Maestro, со счета мобильного телефона, с платежного терминала, в салоне МТС или Связной, через PayPal, WebMoney, Яндекс.Деньги, QIWI Кошелек, бонусными картами или другим удобным Вам способом.