DOCUMENTAÇÃO DO PRATT PARSER

Introdução: O que é um Parser e um Pratt Parser?

Antes de mergulhar no código, é essencial entender o propósito de um **parser** e o que torna o **Pratt Parser** especial.

O que é um Parser?

Um parser é uma componente de software que analisa uma sequência de tokens (gerada por um **lexer**) e organiza esses tokens em uma estrutura lógica, geralmente uma **árvore sintática abstrata (AST)**. No contexto de linguagens de programação, parsers são usados para interpretar expressões matemáticas, comandos de código ou até consultas SQL, garantindo que a sintaxe esteja correta e que a semântica seja compreensível para o computador.

Analogia: Pense no parser como um bibliotecário que recebe uma pilha de palavras soltas (os tokens) e organiza essas palavras em frases e parágrafos coerentes (a AST), seguindo as regras gramaticais de uma língua.

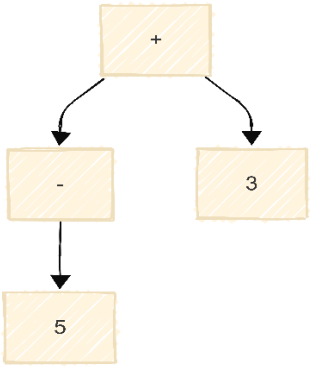
O que é um Pratt Parser?

O **Pratt Parser**, criado por Vaughan Pratt, é uma técnica de **parsing top-down** (de cima para baixo) especialmente eficiente para lidar com expressões que envolvem operadores com diferentes níveis de precedência e associatividade, como em expressões matemáticas. Ele resolve o problema de precedência de operadores de forma elegante, usando o conceito de **binding power** (poder de ligação), que determina a força com que um operador "prende" seus operandos.

Dica: Diferente de outros métodos, como parsers baseados em pilhas ou recursão descendente tradicional, o Pratt Parser é compacto e flexível, permitindo lidar com operadores prefixados (`-5`), infixos (`3 + 4`), e até pós-fixos (`x!`) em uma única estrutura.

Por que usar um Pratt Parser? Ele é ideal para expressões complexas porque: - Simplifica o tratamento de precedência sem precisar de tabelas extensas. - É extensível para novos operadores. - Reduz a complexidade do código em comparação com outros métodos.

No código fornecido, o Pratt Parser é usado para interpretar expressões matemáticas como ***`-5 + 3`***, gerando uma representação estruturada que respeita a precedência e associatividade dos operadores. Veja como ficaria a árvore sintática da expressão na figura abaixo:



Visão Geral do Código

A classe `PrattParser.java` é o coração do parser. Ela recebe uma lista de tokens (produzida pelo **lexer**) e constrói uma árvore de expressões representada pelos modelos `Atom` (valores atômicos, como números ou variáveis) e `Cons` (operações que conectam expressões, como + ou *). O parser utiliza o conceito de **binding power** para decidir a ordem de avaliação das operações.

A Estrutura do Projeto

- **`parser.prat_parser`**: Contém a lógica principal do parser (`PrattParser.java`) e a classe de entrada (`App.java`).
- **`parser.prat_parser.model`**: Define os modelos de dados, como `Expression`, `Atom`, `Cons`, `Token`, `TokenType`, e `BindingPower`.

Antes de discutirmos a classe PrattParser e seus métodos, é importante compreender os três tipos de operadores com os quais esse parser trabalha. Também veremos mais adiante como são atribuídas as precedências a cada um desses operadores.

*1. **Operador Infixo***

É o mais comum: aparece **entre dois operandos**.

Sintaxe:
`A <operador> B`

Exemplos:

```
4 + 5       // Soma
3 * 7       // Multiplicação
10 - 2      // Subtração
a == b      // Comparação
```

No parser:

- Usa **left binding power** (LBP) e **right binding power** (RBP).
- A associatividade (esquerda ou direita) depende da comparação entre esses poderes.

Operador Prefixo

Aparece **antes de um único operando** (unário). Não há nada à esquerda.

Sintaxe:
`<operador> A`

Exemplos:

```
-5          // Menos unário
+3          // Mais unário (raro mas válido)
!true       // Negação lógica
++x         // Pré-incremento
```

No parser:

- Só usa **right binding power** (RBP).
- Define o quanto o operador consome à direita.

3. ***Operador Pós-fixo (Postfix)***

Aparece **depois de um único operando**.

Sintaxe:
`A <operador>`

Exemplos:

```
x++         // Pós-incremento
x--         // Pós-decremento
func()[2]   // Acesso por índice
f!          // Exclamação (hipotética)
```

No parser:

- Só usa **left binding power** (LBP).
- Define o quanto ele pode se "anexar" a expressões já processadas.

Resumo Tabela

Tipo	Exemplo	Forma	Usa LBP?	Usa RBP?
Infixo	`4 + 5`	`A op B`	✅	✅
Prefixo	`-a, !x`	`op A`	❌	✅
Pós-fixo	`x++, x[2]`	`A op`	✅	❌

Análise Detalhada da Classe `PrattParser`

A classe `PrattParser` implementa a lógica do Pratt Parser.

Estrutura da Classe

```
public class PrattParser {
    private int tokenIndex;
    private final List<Token> tokens;

    public PrattParser(Lexer lexer) {
        this.tokens = lexer.tokenize();
        this.tokenIndex = 0;
    }
    // Métodos: nextToken, peekToken, parse, parseExpression, prefixBindingPower, postfixBindingPower, infixBindingPower
}
```

- **Atributos**:
  - `tokenIndex`: Um contador que rastreia a posição atual na lista de tokens.
  - `tokens`: Uma lista imutável de tokens gerada pelo lexer.
- **Construtor**:
  - Recebe um `Lexer`, chama `lexer.tokenize()` para obter os tokens e inicializa `tokenIndex` como 0.

Dica: O uso de uma lista de tokens pré-gerada permite que o parser seja **stateless** (sem estado interno complexo), facilitando a depuração e reutilização.

Métodos Auxiliares: `nextToken` e `peekToken`

```
private Token nextToken() {
//List<Token> tokens = lexer.tokenize();
return tokenIndex < tokens.size() ? tokens.get(tokenIndex++) : new Token(TokenType.EOF, "");
}
private Token peekToken() {
//List<Token> tokens = lexer.tokenize();
return tokenIndex < tokens.size() ? tokens.get(tokenIndex) : new Token(TokenType.EOF, "");
}
```

- **`nextToken()`**: Avança o `tokenIndex` e retorna o token atual. Se não houver mais tokens, retorna um token `EOF` (fim da entrada).
- **`peekToken()`**: Retorna o token atual sem avançar o `tokenIndex`, permitindo "espiar" o próximo token.

Analogia: Pense em `nextToken` como virar a página de um livro, enquanto `peekToken` é apenas olhar a próxima página sem movê-la.

Por que isso é importante? Esses métodos garantem que o parser possa processar tokens sequencialmente e antecipar o próximo token para tomar decisões, essencial para a lógica do Pratt Parser.

Método Principal:

`parse`

```
public Expression parse() {
return parseExpression(0);
}
```

O método `parse` é a entrada pública do parser. Ele simplesmente chama `parseExpression(0)`, iniciando a análise com um **binding power mínimo** de 0, o que permite que qualquer operador seja considerado.

O valor `0` como `minBp` significa que o parser começará sem restrições de precedência, permitindo que todos os operadores sejam avaliados conforme suas regras de binding power.

O Pratt Parser:

- Começa com um número.
- Sempre olha o próximo token para ver **se tem maior precedência** do que o que veio antes.
- Isso permite lidar com **precedência e associatividade** corretamente, **sem precisar escrever gramática recursiva esquerda/direita**.
- Monta a AST usando objetos `Atom` (nós folha) e `Cons(op, [...])` (nós internos com operadores).

O Coração do Pratt Parser: `parseExpression`

O método `parseExpression` é onde a mágica do Pratt Parser acontece. Ele implementa a lógica recursiva para construir a árvore de expressões, respeitando precedência e associatividade.

```
public Expression parseExpression(int minBp) {
    Expression lhs;
    Token token = nextToken();
    // Parte 1: Processa o lado esquerdo (lhs)
    if (token.type() == TokenType.NUMBER || token.type() == TokenType.VARIABLE) {
        lhs = new Atom(token.value());
    } else if (token.type() == TokenType.OPERATOR && token.value().matches("[+-]")) {
        int rBp = prefixBindingPower(token.value().charAt(0));
        Expression rhs = parseExpression(rBp);
        lhs = new Cons(token.value(), List.of(rhs));
    } else if (token.type() == TokenType.LPAREN) {
        lhs = parseExpression(0);
        Token next = nextToken();
        if (next.type() != TokenType.RPAREN) {
            throw new IllegalStateException("Expected ')', found: " + next);
        }
    } else {
        throw new IllegalStateException("Bad token: " + token);
    }
    // Parte 2: Processa operadores infixos e pós-fixos
    while (true) {
        Token opToken = peekToken();
        if (opToken.type() == TokenType.EOF) {
            break;
        }
        if (opToken.type() != TokenType.OPERATOR && opToken.type() != TokenType.LBRACKET) {
            break;
        }
        char op = opToken.value().charAt(0);
        Integer postfixBp = postfixBindingPower(op);
        if (postfixBp != null) {
            if (postfixBp < minBp) {
                break;
            }
            nextToken();
            if (op == '[') {
                Expression rhs = parseExpression(0);
                Token next = nextToken();
                if (next.type() != TokenType.RBRACKET) {
                    throw new IllegalStateException("Expected ']', found: " + next);
                }
                lhs = new Cons(String.valueOf(op), List.of(lhs, rhs));
            } else {
                lhs = new Cons(String.valueOf(op), List.of(lhs));
            }
            continue;
        }
        BindingPower bp = infixBindingPower(op);
        if (bp == null) {
            break;
        }
        int lBp = bp.left();
        int rBp = bp.right();
        if (lBp < minBp) {
            break;
        }
        nextToken();
        if (op == '?') {
            Expression mhs = parseExpression(0);
            Token next = nextToken();
            if (next.type() != TokenType.OPERATOR || !next.value().equals(":")) {
                throw new IllegalStateException("Expected ':', found: " + next);
            }
            Expression rhs = parseExpression(rBp);
            lhs = new Cons(String.valueOf(op), List.of(lhs, mhs, rhs));
        } else {
            Expression rhs = parseExpression(rBp);
            lhs = new Cons(String.valueOf(op), List.of(lhs, rhs));
        }
    }
    return lhs;
}
```

O método `parseExpression` é dividido em duas partes principais:

1. **Processamento do lado esquerdo (lhs)**: Determina o primeiro operando ou subexpressão.

2. **Processamento de operadores infixos e pós-fixos**: Aplica operadores subsequentes, respeitando a precedência via **binding power**.

Parte 1: Processamento do Lado Esquerdo

O parser começa consumindo o primeiro token com `nextToken()` e decide como construir o `lhs` (lado esquerdo da expressão):

- **Se o token for um número ou variável** (`TokenType.NUMBER` ou `TokenType.VARIABLE`):
  - Cria um `Atom` com o valor do token (ex.: `4` ou `x`).
  - **Exemplo**: Para o token `NUMBER(4)`, o `lhs` será `Atom("4")`.
- **Se o token for um operador prefixado (+ ou -)**:
  - Calcula o **binding power** do operador prefixado com `prefixBindingPower`.
  - Faz uma chamada recursiva a `parseExpression(rBp)` para processar o operando à direita (ex.: `-5` → processa `5`).
  - Cria um `Cons` com o operador e o operando (ex.: `Cons("-", [Atom("5")])`).
  - **Exemplo**: Para `-5`, o `lhs` será `Cons("-", [Atom("5")])`.
- **Se o token for um parêntese esquerdo (()**:
  - Faz uma chamada recursiva a `parseExpression(0)` para processar a subexpressão dentro dos parênteses.
  - Verifica se o próximo token é um parêntese direito (`)`), senão lança uma exceção.
  - **Exemplo**: Para `(3 + 4)`, processa `3 + 4` e retorna o resultado como `lhs`.
- **Caso contrário**:
  - Lança uma exceção para tokens inválidos.

Analogia: O `lhs` é como o primeiro ingrediente de uma receita. Ele pode ser algo simples (um número), algo preparado (um operador prefixado com um operando), ou até uma sub-receita (uma expressão entre parênteses).

Parte 2: Processamento de Operadores Infixos e Pós-fixos

Após construir o `lhs`, o parser entra em um loop `while` que verifica se há operadores infixos (ex.: `+`, `*`) ou pós-fixos (ex.: `!`, `[]`) a serem aplicados. O loop continua até que:

- Não haja mais tokens (`EOF`).
- O próximo token não seja um operador ou colchete.
- O **binding power** do operador seja menor que o `minBp`.

Para cada operador encontrado:

- **Operadores Pós-fixos** (`!` ou `[]`):
  - Verifica o **binding power** com `postfixBindingPower`.
  - Se `postfixBp >= minBp`, consome o token e processa o operador.
  - Para `[` (colchete), processa a expressão interna e verifica o `]`.
  - Cria um novo `Cons` com o operador e os operandos.
  - **Exemplo**: Para `x!`, o `lhs` se torna `Cons("!", [Atom("x")])`.
- **Operadores Infixos** (`+`, `-`, `*`, `/`, `=`, `?`, `.`):
  - Obtém o **binding power** com `infixBindingPower`, que retorna um enum `BindingPower` com valores `left` e `right`.
  - Se `lBp >= minBp`, consome o token e processa o lado direito com `parseExpression(rBp)`.
  - Para o operador ternário (`?`), processa a expressão do meio (`mhs`), verifica o `:` e processa o lado direito.
  - Cria um novo `Cons` com o operador e os operandos.
  - **Exemplo**: Para `3 + 4`, o `lhs` se torna `Cons("+", [Atom("3"), Atom("4")])`.

O uso de `minBp` (binding power mínimo) é o que garante a precedência. Operadores com maior binding power (ex.: `*` com `7/8`) são processados antes de operadores com menor binding power (ex.: `+` com `5/6`).

Recursividade em `parseExpression`

O método `parseExpression` é **recursivo** porque:

- Para operadores prefixados, chama a si mesmo para processar o operando direito.
- Para subexpressões entre parênteses, chama a si mesmo com `minBp = 0`.
- Para operadores infixos, chama a si mesmo para processar o lado direito com o `rBp` do operador.

A recursividade **é como abrir uma boneca russa**: cada chamada abre uma nova camada (subexpressão) até chegar ao núcleo (um `Atom`) e depois reconstrói a expressão camada por camada.

Binding Power: O Segredo do Pratt Parser

O conceito de **binding power** é central no Pratt Parser. Ele define a precedência e associatividade dos operadores.

```
        public enum BindingPower {
            PREFIX_PLUS_MINUS(9),
            POSTFIX_EXCL_BRACKET(11),
            INFIX_ASSIGN(2, 1), INFIX_TERNARY(4, 3), INFIX_ADD_SUB(5, 6), INFIX_MUL_DIV(7, 8), INFIX_DOT(14, 13);
            private final int left;
            private final int right;

            // Prefix/Postfix – mesmos pesos
            BindingPower(int bp) {
                this.left = bp;
                this.right = bp;
            }

            // Infix
            BindingPower(int left, int right) {
                this.left = left;
                this.right = right;
            }
            public int left() {
                return left;
            }
            public int right() {
                return right;
            }
        }
```

- Cada operador tem um **binding power** associado, representado por valores `left` e `right`.
- **Prefixados e pós-fixos**: Usam um único valor (ex.: `PREFIX_PLUS_MINUS(9)`).
- **Infixos**: Usam dois valores (`left` para precedência à esquerda, `right` para a próxima recursão).
- **Exemplo**:
  - `INFIX_MUL_DIV(7, 8)`: Multiplicação e divisão têm maior precedência que `INFIX_ADD_SUB(5, 6)` (adição e subtração).
  - `INFIX_ASSIGN(2, 1)`: Atribuição tem baixa precedência, garantindo que outras operações sejam avaliadas antes.

O binding power é como a "força de atração" de um operador. Operadores com maior binding power (ex.: `*`) "puxam" os operandos antes de operadores com menor binding power (ex.: `+`).

```
private int prefixBindingPower(char op) {
    return switch (op) {
        case '+', '-' -> BindingPower.PREFIX_PLUS_MINUS.right();
        default -> throw new IllegalArgumentException("Bad prefix operator: " + op);
    };
}

private Integer postfixBindingPower(char op) {
    return switch (op) {
        case '!', '[' -> BindingPower.POSTFIX_EXCL_BRACKET.left();
        default -> null;
    };
}

private BindingPower infixBindingPower(char op) {
    return switch (op) {
        case '=' -> BindingPower.INFIX_ASSIGN;
        case '?' -> BindingPower.INFIX_TERNARY;
        case '+', '-' -> BindingPower.INFIX_ADD_SUB;
        case '*', '/' -> BindingPower.INFIX_MUL_DIV;
        case '.' -> BindingPower.INFIX_DOT;
        default -> null;
    };
}
```

- **`prefixBindingPower`**: Define o binding power para operadores- **`postfixBindingPower`**: Define o binding power para operadores pós-fixos (`!`, `[`).
- **`infixBindingPower`**: Define o binding power para operadores infixos (`+`, `-`, `*`, `/`, `=`, `?`, `.`).

O uso do operador `switch` com expressões (introduzido no Java 12) torna o código mais limpo e legível, evitando cadeias de `if-else`.

Modelos de Dados: `Expression`, `Atom`, e `Cons`

O resultado do parsing é uma árvore de expressões, representada pela interface selada `Expression`.

```
public sealed interface Expression permits Atom, Cons {
}
```

- **`Atom`**: Representa valores atômicos (números ou variáveis).

  ```
  public record Atom(String value) implements Expression {
      @Override
      public String toString() {
          return value;
      }
  }
  ```

  - **Exemplo**: `Atom("4")` ou `Atom("x")`.
- **`Cons`**: Representa operações com um operador e uma lista de operandos.

  ```
  public record Cons(String operator, List<Expression> operands) implements Expression {
      @Override
      public String toString() {
          if (operands.isEmpty()) {
              return operator;
          }
          return "(" + operator + " " +
          operands.stream()
                  .map(Object::toString)
                  .collect(Collectors.joining(" ")) +
          ")";
      }
  }
  ```

  - **Exemplo**: `Cons("+", [Atom("3"), Atom("4")])` para `3 + 4`.

O uso de **records** (Java 14+) simplifica a criação de classes imutáveis, reduzindo o boilerplate. A interface selada (`sealed`) garante que apenas `Atom` e `Cons` implementem `Expression`.

EXEMPLO PRÁTICO

Expressão a ser analisada:

`-5 + 3`

O que já temos:

- Um **Lexer** que já converteu a expressão em **tokens**, como:

`[OPERATOR(-), NUMBER(5), OPERATOR(+), NUMBER(3), EOF]`

Resultado do Pratt Parser :

Montar uma **árvore de sintaxe abstrata (AST)** que represente a **ordem correta de operações** da expressão. Como os operadores têm diferentes **precedências** (`*` e `/` antes de `+` e `-`), o parser precisa respeitar isso. O resultado final será uma árvore AST:

`(+ (- 5) 3)`

Passo 1: Inicializando o parser com os tokens: [OPERATOR(-), NUMBER(5), OPERATOR(+), NUMBER(3), EOF]

- **Código:**

```
public PrattParser(Lexer lexer) {
    this.tokens = lexer.tokenize();
    this.tokenIndex = 0;
}
```

- **Explicação:** O parser começa recebendo um `Lexer` (um ajudante que quebra a entrada `-5 + 3` em pedaços chamados tokens). Ele converte a entrada em uma lista de tokens: `OPERATOR(-)` (sinal de menos), `NUMBER(5)`, `OPERATOR(+)`, `NUMBER(3)`, e `EOF` (fim da expressão). O índice `tokenIndex` é definido como 0, indicando que começaremos a ler o primeiro token. Pense nisso como o bibliotecário preparando a lista de peças da expressão.

- **Condicional:** Nenhuma condicional neste passo, apenas inicialização.

Passo 2: Iniciando parse da expressão completa

- **Código:**

```
public Expression parse() {
    return parseExpression(0);
}
```

- **Explicação:** O método `parse` é o ponto de entrada para processar a expressão inteira. Ele chama `parseExpression(0)`, começando com a menor prioridade possível (`minBp = 0`), permitindo que todos os operadores sejam considerados. É como o bibliotecário começando a organizar a expressão do zero.

- **Condicional:** Nenhuma condicional, apenas inicia o parsing.

Passo 3: Iniciando parseExpression com precedência mínima (minBp): 0

- **Código:**

```
public Expression parseExpression(int minBp) {
    Expression lhs;
    Token token = nextToken();
    ...
}
```

- **Explicação:** O parser entra no método `parseExpression`, o coração do Pratt Parser. O parâmetro `minBp` (mínima força de ligação) define a prioridade mínima para operadores. Aqui, `minBp = 0` significa que qualquer operador pode ser processado, já que é o início da expressão.

- **Condicional:** Nenhuma condicional, apenas inicia a função.

Passo 4: Consumindo token: OPERATOR(-) (índice atual: 1)

- **Código:**

```
private Token nextToken() {
    return tokenIndex < tokens.size() ? tokens.get(tokenIndex++) : new Token(TokenType.EOF, "");
}
```

- **Explicação:** O parser lê o primeiro token usando `nextToken()`. Como `tokenIndex = 0` e há tokens disponíveis (`tokenIndex < tokens.size()`), ele pega `OPERATOR(-)` e incrementa `tokenIndex` para 1. É como o bibliotecário pegando a primeira peça da expressão.

- **Condicional:**
  - `tokenIndex < tokens.size()`: `true`, porque `tokenIndex = 0` e `tokens.size() = 5`. Isso retorna `OPERATOR(-)` em vez de `EOF`.

Passo 5: Processando token inicial: OPERATOR(-)

- **Código:**

```
Token token = nextToken();
```

- **Explicação:** O parser armazena o token `OPERATOR(-)` em `token` para processá-lo. Este é o primeiro pedaço da expressão `-5 + 3`, indicando um operador prefixado (sinal de menos antes de um número).

- **Condicional:** Nenhuma condicional, apenas lê o token.

Passo 6: Calculando binding power para operador prefixado -: 9

- **Código:**

```
private int prefixBindingPower(char op) {
    return switch (op) {
        case '+', '-' -> BindingPower.PREFIX_PLUS_MINUS.right();
        default -> throw new IllegalArgumentException("Bad prefix operator: " + op);
    };
}
```

- **Explicação:** Como o token é `OPERATOR(-)`, o parser verifica se é um operador prefixado. Ele chama `prefixBindingPower('-')`, que retorna o *binding power* à direita (9) para o operador `-`. O *binding power* é a "força" do operador, indicando sua prioridade.

- **Condicional:**

  - `switch (op)`: O operador `-` está na lista `('+', '-')`, então retorna `BindingPower.PREFIX_PLUS_MINUS.right() = 9`. Não cai no `default`, evitando a exceção.

Passo 7: Encontrado operador prefixado: -, binding power à direita: 9

- **Código:**

```
} else if (token.type() == TokenType.OPERATOR && token.value().matches("[+-]")) {
    int rBp = prefixBindingPower(token.value().charAt(0));
    ...
}
```

- **Explicação:** O parser confirma que o token `OPERATOR(-)` é um operador prefixado, porque `token.type() == TokenType.OPERATOR` e `token.value()` corresponde ao padrão `[+-]`. Ele obtém o *binding power* à direita (9).

- **Condicional:**

  - `token.type() == TokenType.OPERATOR`: `true`, porque o token é `OPERATOR(-)`.
  - `token.value().matches("[+-]")`: `true`, porque `-` está no padrão `[+-]` (expressão regular para `+` ou `-`).
  - As condições anteriores (`token.type() == TokenType.NUMBER || token.type() == TokenType.VARIABLE` e `token.type() == TokenType.LPAREN`) são falsas, então o parser entra neste bloco `else if`.

Passo 8: Iniciando parseExpression com precedência mínima (minBp): 9

- **Código:**

```
Expression rhs = parseExpression(rBp);
```

- **Explicação:** O parser chama `parseExpression(9)` para processar o que vem após o operador prefixado `-` (neste caso, o número 5). O `minBp = 9` significa que só operadores com prioridade maior ou igual a 9 serão considerados nesta chamada recursiva. É como o bibliotecário pedindo a um colega para organizar a parte após o `-`.

- **Condicional:** Nenhuma condicional, apenas inicia a recursão.

Passo 9: Consumindo token: NUMBER(5) (índice atual: 2)

- **Código:** (mesmo que Passo 4, método `nextToken`)
- **Explicação:** Na nova chamada de `parseExpression(9)`, o parser lê o próximo token, que é `NUMBER(5)`. O `tokenIndex` avança para 2.
- **Condicional:**
  - `tokenIndex < tokens.size()`: `true`, porque `tokenIndex = 1` e `tokens.size() = 5`.

Passo 10: Processando token inicial: NUMBER(5)

- **Código:** (mesmo que Passo 5)
- **Explicação:** O parser processa o token `NUMBER(5)` na chamada recursiva de `parseExpression(9)`.
- **Condicional:** Nenhuma condicional.

Passo 11: Encontrado átomo (número ou variável): 5, criando nó: 5

- **Código:**

```
if (token.type() == TokenType.NUMBER || token.type() == TokenType.VARIABLE) {
    lhs = new Atom(token.value());
}
```

- **Explicação:** O token `NUMBER(5)` é um número, então o parser cria um nó `Atom(5)`, uma "folha" na árvore da expressão. É como o bibliotecário anotando o número 5 como uma peça básica.
- **Condicional:**
  - `token.type() == TokenType.NUMBER || token.type() == TokenType.VARIABLE`: `true`, porque `token.type() == TokenType.NUMBER`. Isso faz o parser entrar neste bloco `if`, ignorando os outros `else if`.

Passo 12: Espiando token (sem consumir): OPERATOR(+)

- **Código:**

```
Token opToken = peekToken();
```

- **Explicação:** O parser verifica o próximo token sem consumi-lo, usando `peekToken()`. Ele vê `OPERATOR(+)`, indicando que pode haver um operador binário após 5.
- **Condicional:**
  - `tokenIndex < tokens.size()`: `true`, porque `tokenIndex = 2` e `tokens.size() = 5`.

Passo 13: Verificando próximo token para operador: OPERATOR(+)

- **Código:**

```
Token opToken = peekToken();
```

- **Explicação:** O parser verifica se `OPERATOR(+)` é um operador válido para continuar a expressão.
- **Condicional:** Nenhuma condicional direta, mas o resultado de `peekToken()` depende da condicional do método.

Passo 14: Operador identificado: +

- **Código:**

```
char op = opToken.value().charAt(0);
```

- **Explicação:** O parser extrai o caractere + do token para processá-lo como operador.
- **Condicional:** Nenhuma condicional.

Passo 15: Calculando binding power para operador pós-fixado +: nenhum

- **Código:**

```
Integer postfixBp = postfixBindingPower(op);
```

- **Explicação:** O parser verifica se + é um operador pós-fixado (como `!` ou `[`). Como + não está na lista (`'!', '['`), `postfixBindingPower` retorna `null`.
- **Condicional:**
  - `switch (op)`: O operador + não está na lista, então cai no `default` e retorna `null`.

Passo 16: Calculando binding power para operador infixo +: INFIX_ADD_SUB

- **Código:**

```
BindingPower bp = infixBindingPower(op);
```

- **Explicação:** Como + não é pós-fixado, o parser verifica se é infixo. O `infixBindingPower('+')` retorna `BindingPower.INFIX_ADD_SUB`, com `left = 5` e `right = 6`.
- **Condicional:**
  - `switch (op)`: O operador + está na lista, então retorna `BindingPower.INFIX_ADD_SUB`.
  -

Passo 17: Operador infixo encontrado: +, left BP: 5, right BP: 6

- **Código:**

```
int lBp = bp.left();
int rBp = bp.right();
```

- **Explicação:** O parser obtém os *binding powers* do +: `left = 5` (prioridade à esquerda) e `right = 6` (prioridade à direita).
- **Condicional:** Nenhuma condicional, apenas extrai os valores.

Passo 18: Binding power à esquerda (5) menor que minBp (9), encerrando loop

- **Código:**

```
if (lBp < minBp) {
    break;
}
```

- **Explicação:** Na chamada recursiva (`parseExpression(9)`), o parser compara o *binding power* à esquerda do + (`lBp = 5`) com o `minBp = 9`. Como `5 < 9`, o parser não processa o +, porque o operador atual (- prefixado) tem maior prioridade. Isso encerra o loop, retornando `Atom(5)`.
- **Condicional:**
  - `lBp < minBp`: `true`, porque `lBp = 5` e `minBp = 9`. Isso faz o parser sair do loop `while`, respeitando a precedência do operador prefixado -.

Passo 19: Finalizando parseExpression, retornando: 5

- **Código:**

```
return lhs;
```

- **Explicação:** A chamada recursiva de `parseExpression(9)` termina, retornando `Atom(5)` como o lado direito do operador prefixado -.
- **Condicional:** Nenhuma condicional.

Passo 20: Após parse recursivo, lado direito: 5

- **Código:**

```
Expression rhs = parseExpression(rBp);
```

- **Explicação:** O parser volta à chamada original, com o lado direito do - prefixado sendo `Atom(5)`.
- **Condicional:** Nenhuma condicional.

Passo 21: Criando nó para operador prefixado: (- 5)

- **Código:**

```
lhs = new Cons(token.value(), List.of(rhs));
```

- **Explicação:** O parser cria um nó `Cons("-", [Atom(5)])` para representar `-5`. Este é um galho na árvore que conecta o operador - ao número 5.
- **Condicional:** Nenhuma condicional.

Passo 22: Espiando token (sem consumir): OPERATOR(+)

- **Código:**

```
Token opToken = peekToken();
```

- **Explicação:** Na chamada principal (`parseExpression(0)`), o parser verifica o próximo token, que é `OPERATOR(+)`.
- **Condicional:**
  - `tokenIndex < tokens.size()`: `true`, porque `tokenIndex = 2` e `tokens.size() = 5`.

Passo 23: Verificando próximo token para operador: OPERATOR(+)

- **Código:** (mesmo que Passo 22)
- **Explicação:** O parser verifica se `OPERATOR(+)` é um operador válido.
- **Condicional:** Nenhuma condicional direta.

Passo 24: Operador identificado: +

- **Código:**

```
char op = opToken.value().charAt(0);
```

- **Explicação:** O parser extrai o + para processá-lo.
- **Condicional:** Nenhuma condicional.

Passo 25: Calculando binding power para operador pós-fixado +: nenhum

- **Código:**

```
Integer postfixBp = postfixBindingPower(op);
```

- **Explicação:** O parser verifica se + é pós-fixado, retornando `null`.
- **Condicional:**
  - `switch (op)`: + não está na lista, retorna `null`.

Passo 26: Calculando binding power para operador infixo +: INFIX_ADD_SUB

- **Código:**

```
BindingPower bp = infixBindingPower(op);
```

- **Explicação:** O parser confirma que + é infixo, retornando `BindingPower.INFIX_ADD_SUB`.
- **Condicional:**
  - `switch (op)`: + está na lista, retorna `BindingPower.INFIX_ADD_SUB`.

Passo 27: Operador infixo encontrado: +, left BP: 5, right BP: 6

- **Código:**

```
int lBp = bp.left();
int rBp = bp.right();
```

- **Explicação:** O parser obtém os *binding powers* do +: `left = 5`, `right = 6`.
- **Condicional:** Nenhuma condicional.

Passo 28: Binding power à esquerda (5) maior ou igual a minBp (0), continuando loop

- **Código:**

```
if (lBp < minBp) {
    break;
}
```

- **Explicação:** Na chamada principal (`parseExpression(0)`), o parser compara o *binding power* à esquerda do + (`lBp = 5`) com o `minBp = 0`. Como `5 >= 0`, o parser decide processar o operador +, pois sua precedência é suficiente. Isso permite que o loop `while` continue, avançando para consumir o token. É como o bibliotecário decidindo que a peça + pode ser usada agora, já que tem força suficiente.
- **Condicional:**
  - `lBp < minBp`: `false`, porque `lBp = 5` e `minBp = 0`. Isso faz o parser continuar o loop.

Passo 29: Consumindo token: OPERATOR(+) (índice atual: 3)

- **Código:**

```
nextToken();
```

- **Explicação:** Após confirmar que o operador + pode ser processado, o parser consome o token `OPERATOR(+)`, avançando `tokenIndex` para 3.
- **Condicional:** Nenhuma condicional direta, mas depende da condicional anterior (`lBp < minBp` ser `false`).

Passo 30: Verificando se o operador é ternário: +, não é ternário, tratando como binário

- **Código:**

```
if (op == '?') {
    Expression mhs = parseExpression(0);
    Token next = nextToken();
    if (next.type() != TokenType.OPERATOR || !next.value().equals(":")) {
        throw new IllegalStateException("Expected ':', found: " + next);
    }
    Expression rhs = parseExpression(rBp);
    lhs = new Cons(String.valueOf(op), List.of(lhs, mhs, rhs));
} else {
    Expression rhs = parseExpression(rBp);
    lhs = new Cons(String.valueOf(op), List.of(lhs, rhs));
}
```

- **Explicação:** O parser verifica se o operador é o ternário `?` (usado em expressões como `a ? b : c`). Como o operador atual é `+`, a condição `op == '?'` é falsa, e o parser entra no bloco `else`, tratando o `+` como um operador binário comum. Isso significa que o parser processará o lado direito da expressão (neste caso, 3) como parte de uma operação binária. É como o bibliotecário vendo que a peça + não é uma escolha complexa (ternária) e decidindo organizá-la como uma simples adição.

- **Condicional:**

  - `op == '?'`: `false`, porque `op = '+'`. Isso faz o parser entrar no bloco `else`, onde processará o operador como binário.

Passo 31: Iniciando parse de operador binário: +

- **Código:**

```
Expression rhs = parseExpression(rBp);
```

- **Explicação:** O parser começa a processar o `+` como um operador binário, chamando `parseExpression(6)` para processar o lado direito (número 3). O `rBp = 6` reflete a prioridade à direita do `+`.

- **Condicional:** Nenhuma condicional.

Passo 32: Iniciando parseExpression com precedência mínima (minBp): 6

- **Código:** (mesmo que Passo 8)
- **Explicação:** O parser inicia `parseExpression(6)` para processar o lado direito do `+`. O `minBp = 6` garante que apenas operadores com precedência maior ou igual a 6 sejam considerados.
- **Condicional:** Nenhuma condicional.

Passo 33: Consumindo token: NUMBER(3) (índice atual: 4)

- **Código:** (mesmo que Passo 4)
- **Explicação:** O parser lê `NUMBER(3)`, avançando `tokenIndex` para 4.
- **Condicional:**
  - `tokenIndex < tokens.size()`: `true`, porque `tokenIndex = 3` e `tokens.size() = 5`.

Passo 34: Processando token inicial: NUMBER(3)

- **Código:** (mesmo que Passo 5)
- **Explicação:** O parser processa `NUMBER(3)`.
- **Condicional:** Nenhuma condicional.

Passo 35: Encontrado átomo (número ou variável): 3, criando nó: 3

- **Código:** (mesmo que Passo 11)
- **Explicação:** O parser cria um `Atom(3)` para o número 3.
- **Condicional:**
  - `token.type() == TokenType.NUMBER || token.type() == TokenType.VARIABLE`: `true`, porque `token.type() == TokenType.NUMBER`.

Passo 36: Espiando token (sem consumir): EOF

- **Código:** (mesmo que Passo 12)
- **Explicação:** O parser verifica o próximo token e encontra `EOF`, indicando o fim da expressão.
- **Condicional:**
  - `tokenIndex < tokens.size()`: `false`, porque `tokenIndex = 4` e `tokens.size() = 5`, mas o último token é `EOF`.

Passo 37: Verificando próximo token para operador: EOF

- **Código:** (mesmo que Passo 13)
- **Explicação:** O parser verifica que o próximo token é `EOF`.
- **Condicional:** Nenhuma condicional direta.

Passo 38: Fim da expressão (EOF), encerrando loop

- **Código:**

```
if (opToken.type() == TokenType.EOF) {
    break;
}
```

- **Explicação:** Como o token é `EOF`, o parser encerra o loop `while`, pois não há mais operadores para processar.
- **Condicional:**
  - `opToken.type() == TokenType.EOF`: `true`, porque o token é `EOF`, fazendo o parser sair do loop.
  -

Passo 39: Finalizando parseExpression, retornando: 3

- **Código:** (mesmo que Passo 19)
- **Explicação:** A chamada `parseExpression(6)` retorna `Atom(3)`.
- **Condicional:** Nenhuma condicional.

Passo 40: Lado direito parseado: 3

- **Código:** (mesmo que Passo 20)
- **Explicação:** O parser tem o lado direito do `+` como `Atom(3)`.
- **Condicional:** Nenhuma condicional.

Passo 41: Criando nó para operador binário: (+ (- 5) 3)

- **Código:**

```
lhs = new Cons(String.valueOf(op), List.of(lhs, rhs));
```

- **Explicação:** O parser cria o nó final `Cons("+", [Cons("-", [Atom(5)]), Atom(3)])` para representar `-5 + 3`.
- **Condicional:** Nenhuma condicional.

Passo 42: Espiando token (sem consumir): EOF

- **Código:** (mesmo que Passo 12)
- **Explicação:** O parser verifica novamente o próximo token e encontra `EOF`.
- **Condicional:**
  - `tokenIndex < tokens.size()`: `false`, retorna `EOF`.
  -

Passo 43: Verificando próximo token para operador: EOF

- **Código:** (mesmo que Passo 13)
- **Explicação:** O parser verifica `EOF`.
- **Condicional:** Nenhuma condicional direta.
-

Passo 44: Fim da expressão (EOF), encerrando loop

- **Código:** (mesmo que Passo 38)
- **Explicação:** O parser encerra o loop principal, pois `EOF` indica o fim da expressão.
- **Condicional:**
  - `opToken.type() == TokenType.EOF`: `true`.

Passo 45: Finalizando parseExpression, retornando: (+ (- 5) 3)

- **Código:**

```
return lhs;
```

- **Explicação:** A chamada principal `parseExpression(0)` retorna a árvore final `Cons("+", [Cons("-", [Atom(5)]), Atom(3)])`. Neste ponto, o parser terminou de construir a árvore da expressão `-5 + 3`, que agora está organizada como uma

adição binária (`+`) entre o operador prefixado `-5` (representado por `Cons("-", [Atom(5)])`) e o número `3` (representado por `Atom(3)`). É como o bibliotecário entregando o livro finalizado, com todas as peças organizadas na ordem correta.

- **Condicional:** Nenhuma condicional.