Рекурсивный спуск

Лекция №5

Задача: разобрать и вычислить выражения

```
SELECT (4 + (5 / 4)) * 2;

SELECT 2, 7 + 1.5, 10.25;

SELECT GREATEST(7, 10 - 4);

SELECT LEAST(10, 17 / 2);
```

Форма записи выражений

Формы записи бинарных операций

Эквивалентные формы записи:

- `4 + 5 / 4` — инфиксная

Формы записи бинарных операций

Эквивалентные формы записи:

- 4 + 5 / 4 — инфиксная
- + 4 / 5 4 — префиксная (польская)

Формы записи бинарных операций

Эквивалентные формы записи:

- 4 + 5 / 4 — инфиксная
- + 4 / 5 4 — префиксная (польская)
- 4 5 4 / + — постфиксная (обратная польская)

Формы записи бинарных операций

Эквивалентные формы записи:

- `4 + 5 / 4` — инфиксная
- `+ 4 / 5 4` — префиксная (польская)
- `4 5 4 / +` — постфиксная (обратная польская)

Человек привык читать инфиксную форму

Формы записи бинарных операций

Эквивалентные формы записи:

- `4 + 5 / 4` — инфиксная
- `+ 4 / 5 4` — префиксная (польская)
- `4 5 4 / +` — постфиксная (обратная польская)

Другое выражение:

- `(4 + 5) / 4` — инфиксная
- `/ + 4 5 4` — префиксная (польская)
- `4 5 + 4 /` — постфиксная (обратная польская)

Формы записи бинарных операций

Эквивалентные формы записи:

- `4 + 5 / 4` — инфиксная
- `+ 4 / 5 4` — префиксная (польская)
- `4 5 4 / +` — постфиксная (обратная польская)

Другое выражение:

- `(4 + 5) / 4` — инфиксная
- `/ + 4 5 4` — префиксная (польская)
- `4 5 + 4 /` — постфиксная (обратная польская)

Скобки нужны только для инфиксной формы

Приоритет операций (Operator precedence)

Таблица приоритетов

1	`a ** b`	возведение в степень
2	`+a, -a`	унарные “плюс” и “минус”
3	`a * b, a / b`	умножение, деление
4	`a + b, a - b`	сложение, вычитание

Таблица приоритетов

1	`a ** b`	возведение в степень
2	`+a, -a`	унарные “плюс” и “минус”
3	`a * b, a / b`	умножение, деление
4	`a + b, a - b`	сложение, вычитание

```
8 + 7 + 3 → ((8 + 7) + 3)
8 + 7 * 3 → (8 + (7 * 3))
```

Таблица приоритетов в реальных языках

- C++ — 17 уровней приоритетов
- Python — 18 уровней приоритетов
- Go — 5 уровней приоритетов

5	* / % << >> & &^
4	+ - | ^
3	== != < <= > >=
2	&&
1	||

Языки LISP и Clojure

Все операции — в префиксной нотации

```
(+ 4 (/ 5 3))
```

Ассоциативность операций

Ассоциативность операций

Левоассоциативные

```
2 - 8 + 7
```

→ `(2 - 8) + 7`

→ `1`

Ассоциативность операций

Левоассоциативные

```
2 - 8 + 7
```

→ `(2 - 8) + 7`

→ `1`

Изменение порядка изменит результат:

```
2 - (8 + 7)
```

→ `-13`

Ассоциативность операций

Левоассоциативные

```
2 - 8 + 7
```

→ `(2 - 8) + 7`

→ `1`

Правоассоциативные

```
2 ** 3 ** 2
```

→ `2 ** (3 ** 2)`

→ `512`

Изменение порядка изменит результат:

```
2 - (8 + 7)
```

→ `-13`

Вычисление выражений через постфиксную форму

Вычисление выражений с помощью стека

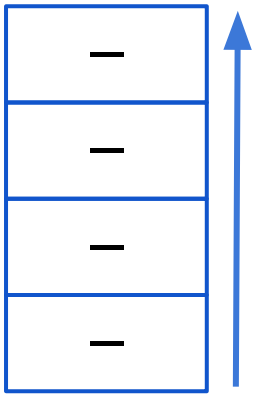
1. Преобразуем в постфиксную нотацию
   - 4 + 5 / 4 → 4 5 4 / +
2. Вычисляем с помощью стека
   - Число → push
   - Бинарная операция → pop, pop, push

Вычисление выражений с помощью стека

1. Преобразуем в постфиксную нотацию
   - 4 + 5 / 4 → 4 5 4 / +
2. Вычисляем с помощью стека
   - Число → push
   - Бинарная операция → pop, pop, push

Выражение: 4 5 4 / +

Стек:

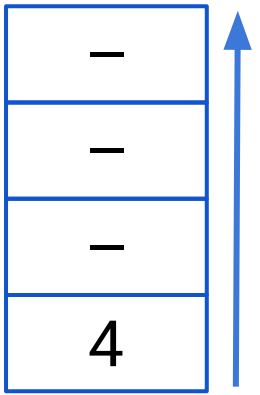


Вычисление выражений с помощью стека

1. Преобразуем в постфиксную нотацию
   - 4 + 5 / 4 → 4 5 4 / +
2. Вычисляем с помощью стека
   - Число → push
   - Бинарная операция → pop, pop, push

Выражение: 4 5 4 / +

Стек:

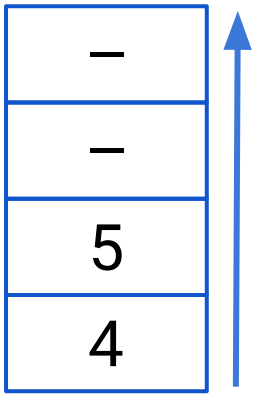


Вычисление выражений с помощью стека

1. Преобразуем в постфиксную нотацию
   - 4 + 5 / 4 → 4 5 4 / +
2. Вычисляем с помощью стека
   - Число → push
   - Бинарная операция → pop, pop, push

Выражение: 4 5 4 / +

Стек:

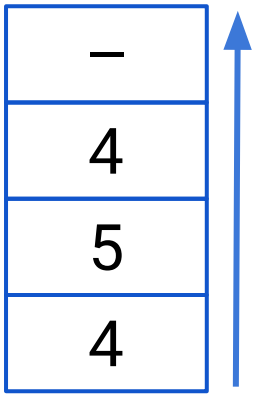


Вычисление выражений с помощью стека

1. Преобразуем в постфиксную нотацию
   - 4 + 5 / 4 → 4 5 4 / +
2. Вычисляем с помощью стека
   - Число → push
   - Бинарная операция → pop, pop, push

Выражение: 4 5 4 / +

Стек:

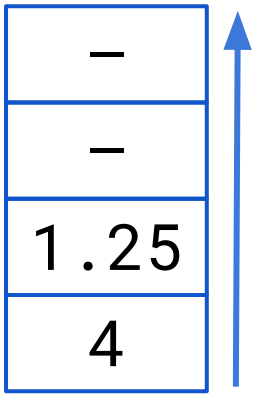


Вычисление выражений с помощью стека

1. Преобразуем в постфиксную нотацию
   - 4 + 5 / 4 → 4 5 4 / +
2. Вычисляем с помощью стека
   - Число → push
   - Бинарная операция → pop, pop, push

Выражение: 4 5 4 / +

Стек:

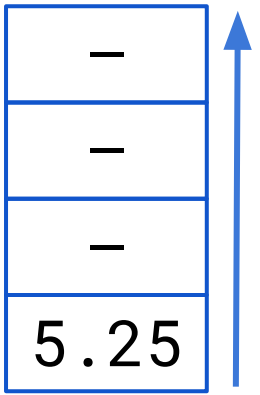


Вычисление выражений с помощью стека

1. Преобразуем в постфиксную нотацию
   - 4 + 5 / 4 → 4 5 4 / +
2. Вычисляем с помощью стека
   - Число → push
   - Бинарная операция → pop, pop, push

Выражение: 4 5 4 / +

Стек:



Алгоритм сортировочной станции (Shunting Yard)

1. Преобразует инфиксную запись в постфиксную
   - 4 5 4 / +
2. Низкий расход памяти
   - стек для состояния
   - очередь для результатов
3. Недостатки
   - Неустойчив к ошибкам синтаксиса
   - Разбирает только выражения

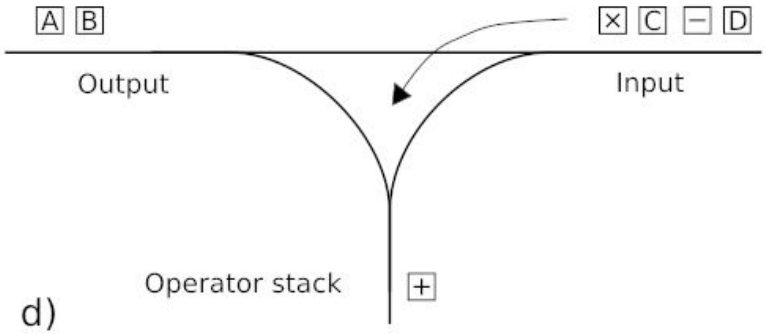


иллюстрация из Wikipedia

Дерево разбора

Пример входных данных

```
SELECT 2, 7 + 1.5
```

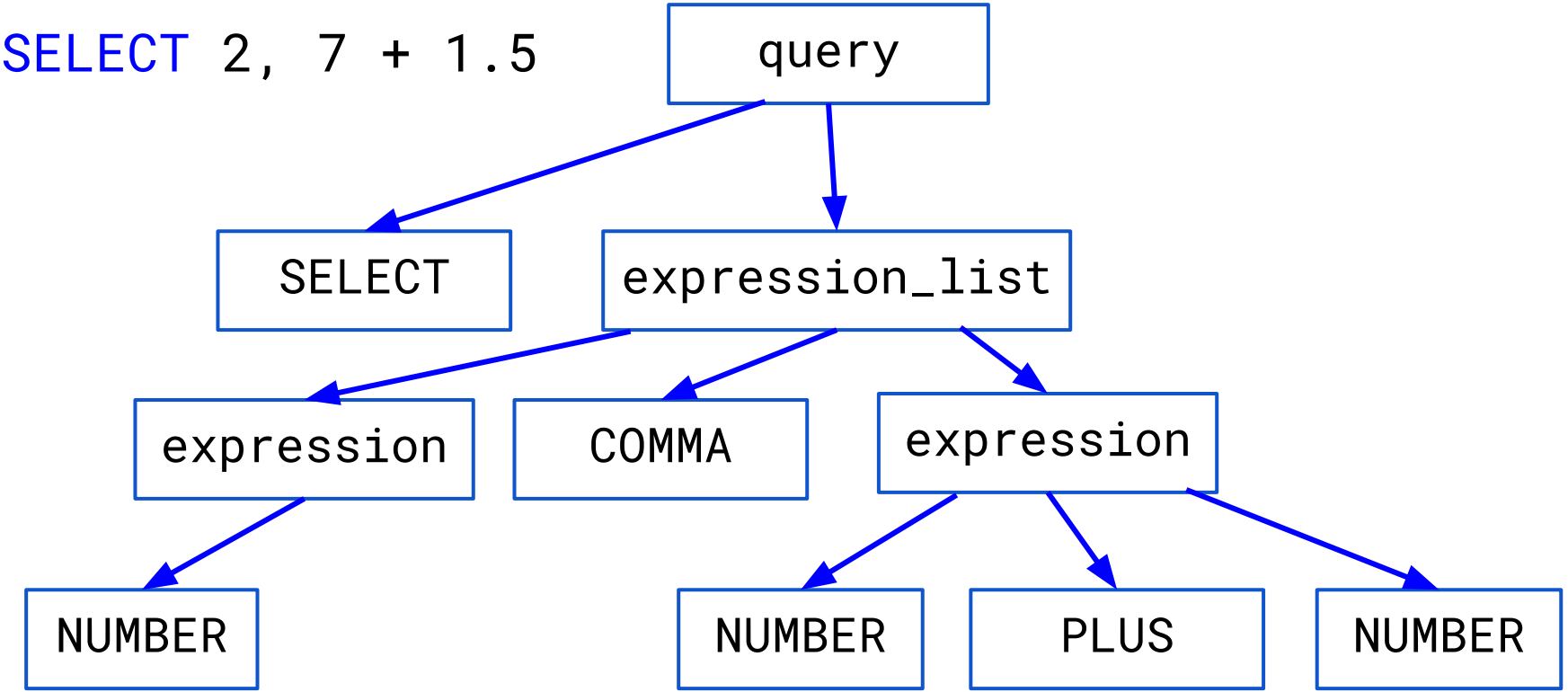
Минимальная грамматика

```
SELECT 2, 7 + 1.5
```

```
query = "SELECT", expression_list ;
expression_list = expression, { ",", expression } ;
expression = NUMBER, { "+", NUMBER } ;
```

Дерево разбора

```
SELECT 2, 7 + 1.5
```

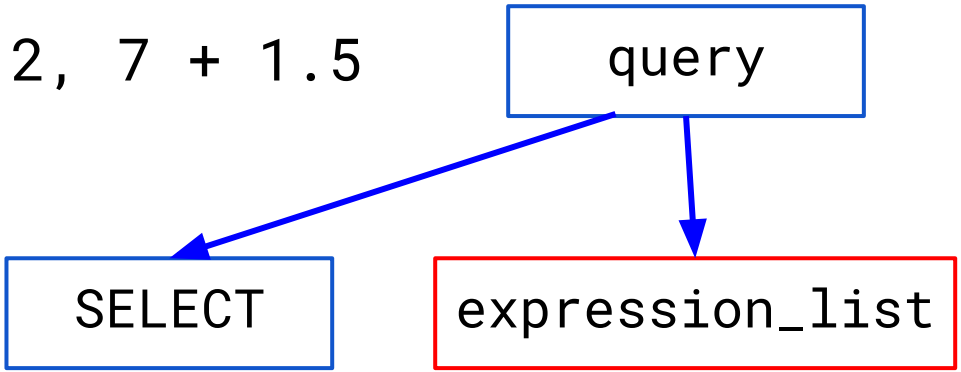


Разбор сверху вниз (top-down parsing)

```
SELECT 2, 7 + 1.5
```

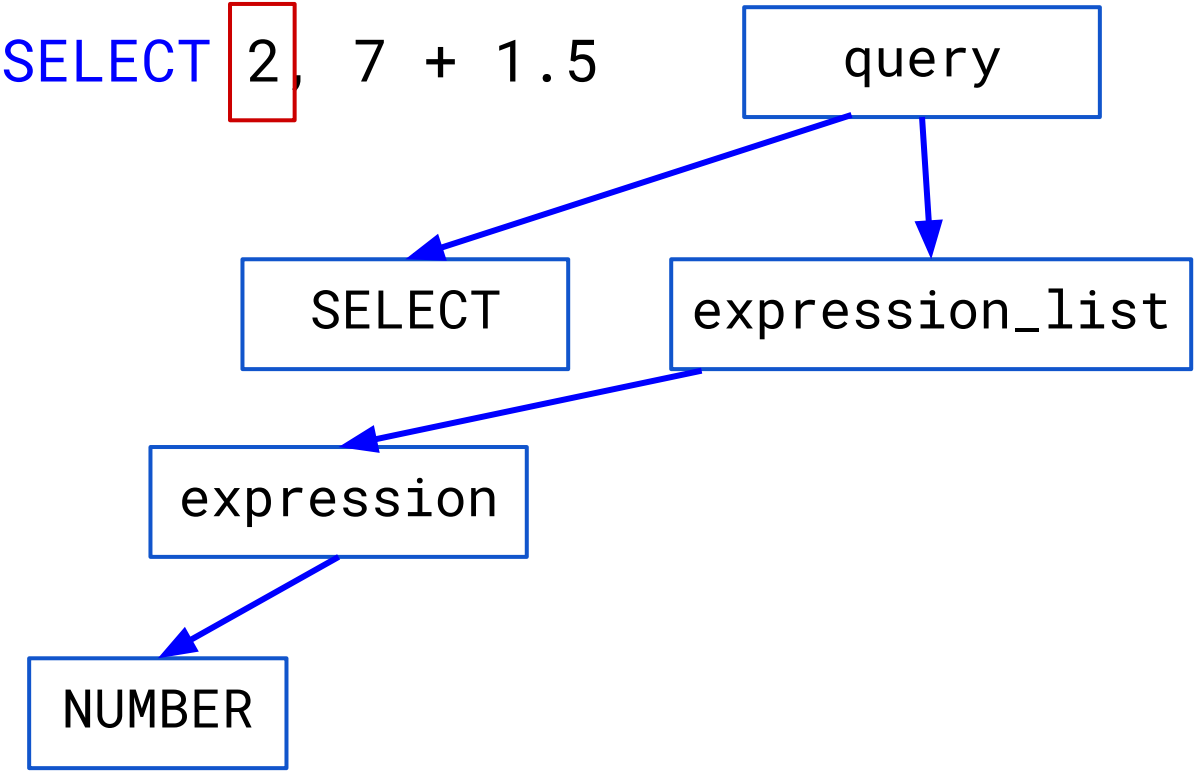
Разбор сверху вниз

```
SELECT 2, 7 + 1.5
```



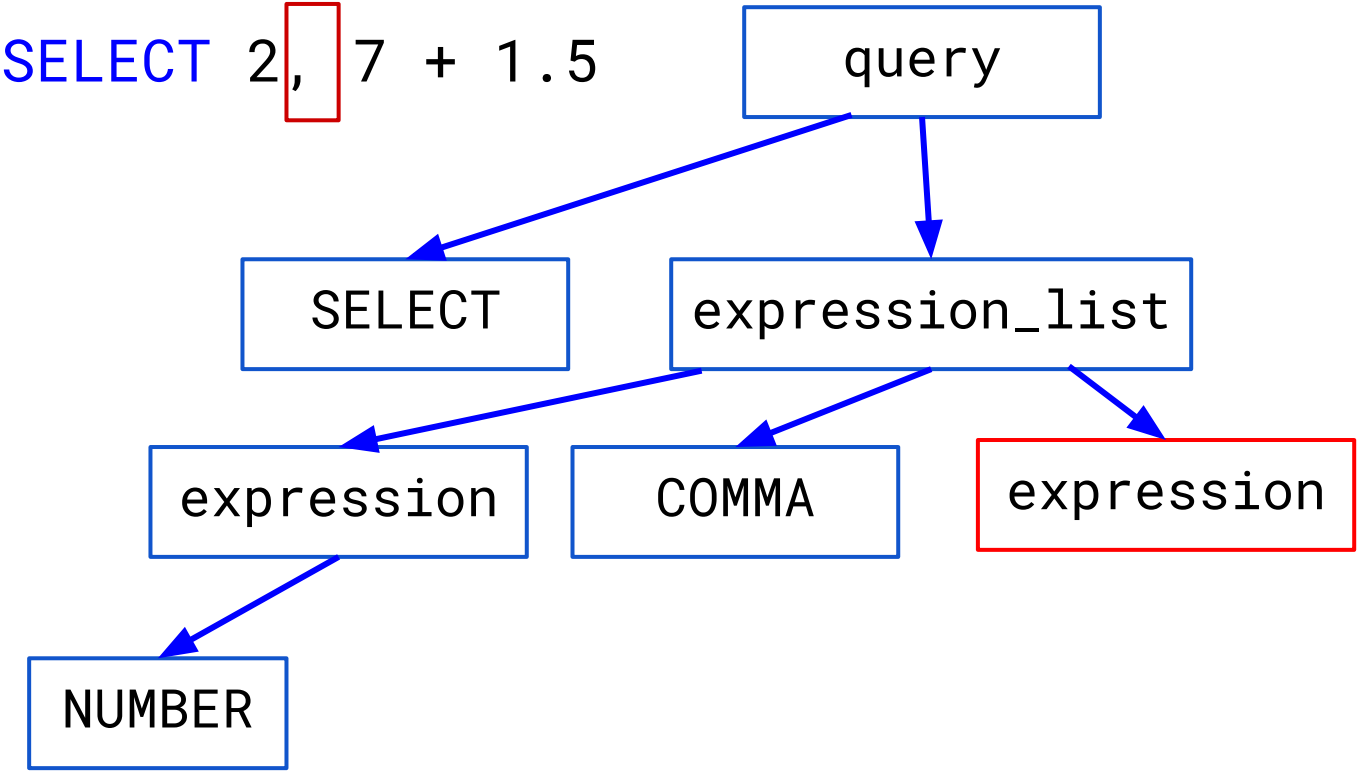
Разбор сверху вниз

```
SELECT 2, 7 + 1.5
```



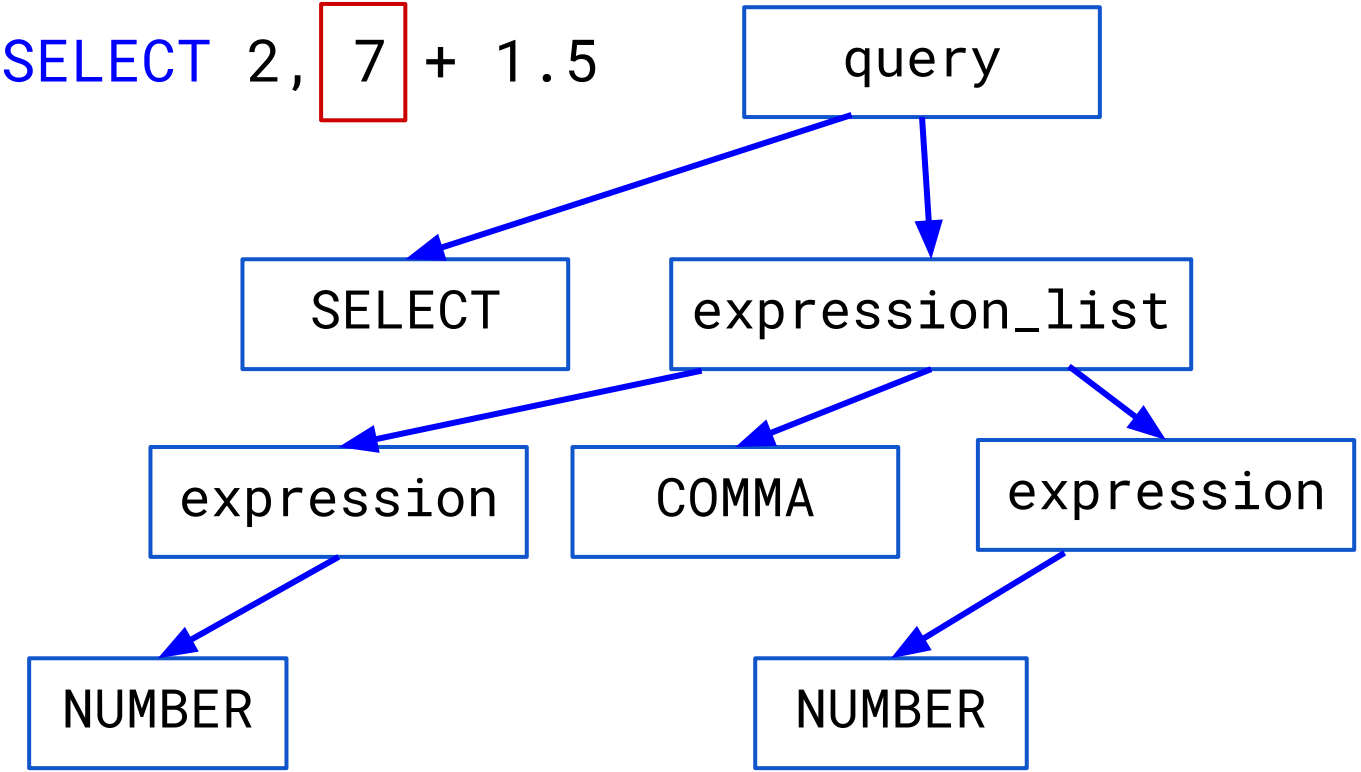
Разбор сверху вниз

```
SELECT 2, 7 + 1.5
```



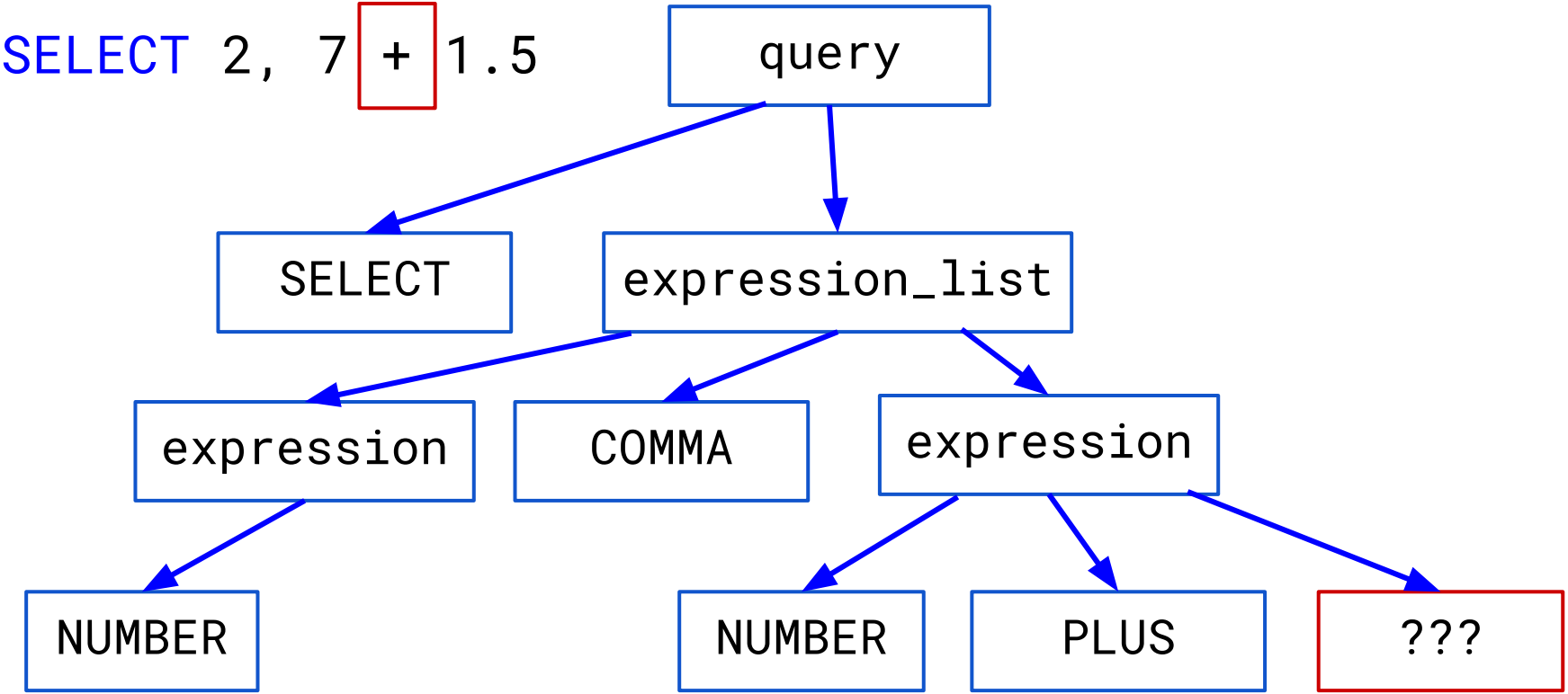
Разбор сверху вниз

```
SELECT 2, 7 + 1.5
```



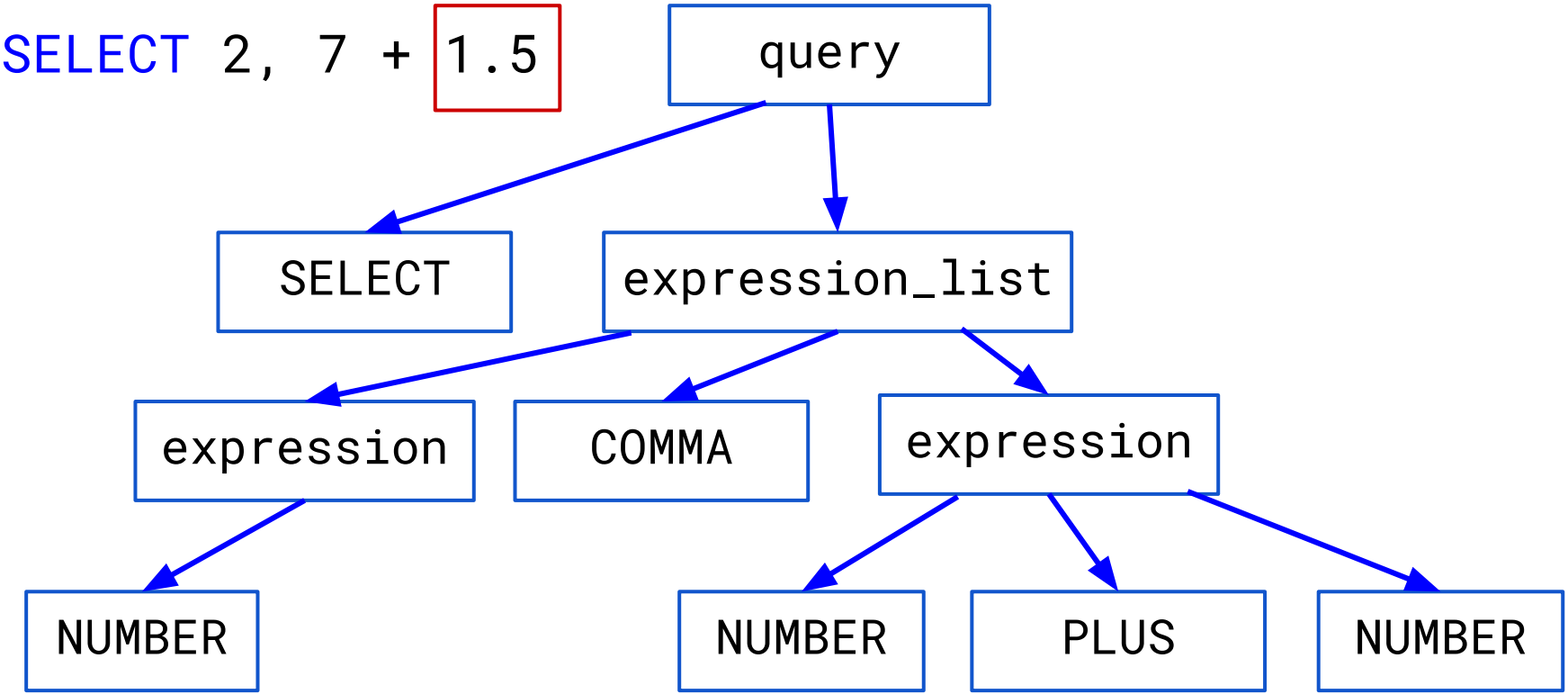
Разбор сверху вниз

```
SELECT 2, 7 + 1.5
```

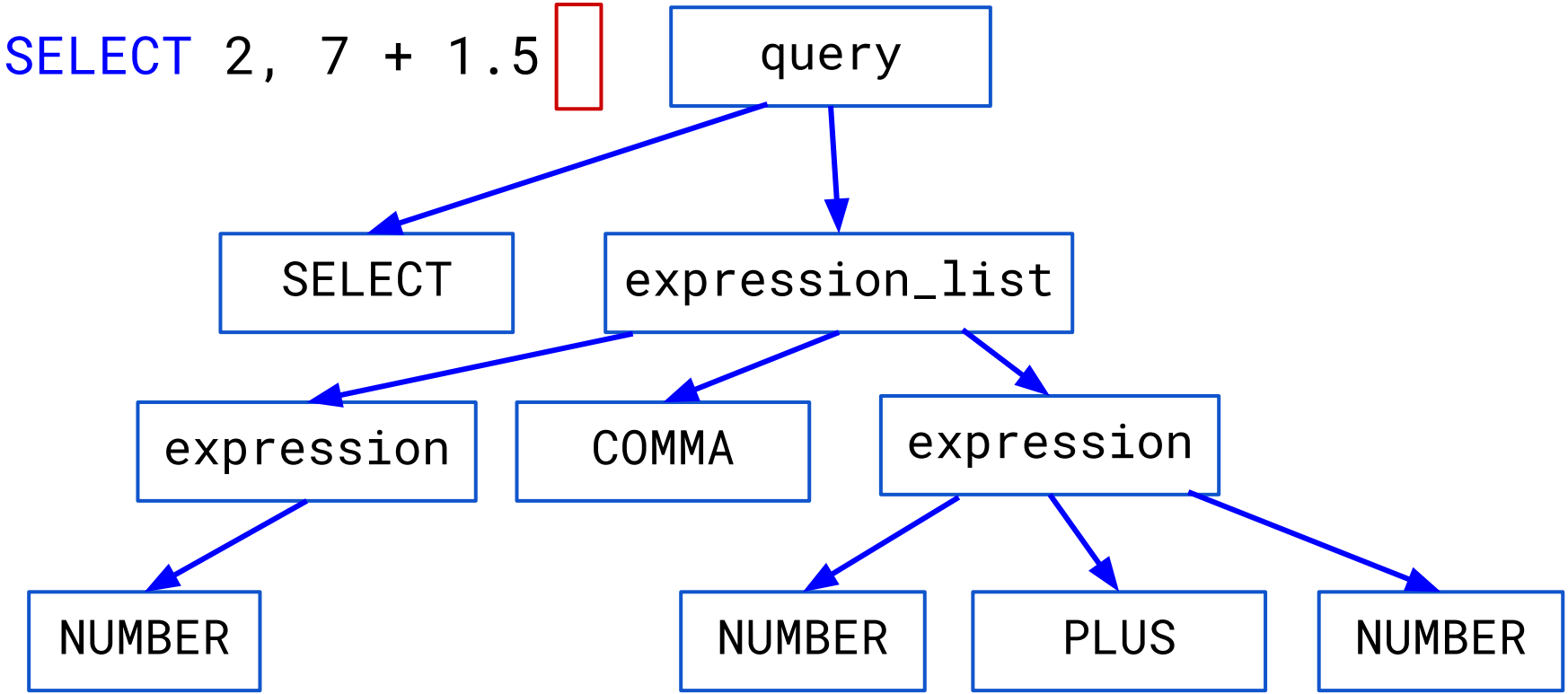


Разбор сверху вниз

```
SELECT 2, 7 + 1.5
```



Разбор сверху вниз



Подготовка грамматики к рекурсивному спуску

Устранение левой рекурсии

EBNF с левой рекурсией:

```
expr = expr, [("+" | "-"), term];

term = term, [("*" | "/"), factor];

factor = identifier | number
```

Устранение левой рекурсии

EBNF с левой рекурсией:

```
expr = expr, [("+" | "-"), term];

term = term, [("*" | "/"), factor];

factor = identifier | number
```

Для рекурсивного спуска левая рекурсия вызывает бесконечную рекурсию

Устранение левой рекурсии

Замена на правую рекурсию:

```
expr = term, [("+" | "-"), expr];

term = factor, [("*" | "/"), term];

factor = identifier | number
```

Устранение левой рекурсии

Замена на оператор повтора:

```
expr = term, {("+" | "-"), term};

term = factor, {("*" | "/"), factor};

factor = identifier | number
```

Факторизация

Два правила с одинаковым префиксом:

```
factor = variable | func_call | number ;

func_call = identifier, "(", expression_list, ")" ;

variable = identifier ;
```

Факторизация

Два правила с одинаковым префиксом:

```
factor = variable | func_call | number ;

func_call = identifier, "(", expression_list, ")" ;

variable = identifier ;
```

Для рекурсивного спуска левая факторизация создаёт неоднозначность

Факторизация

Сводим общий префикс в одно правило:

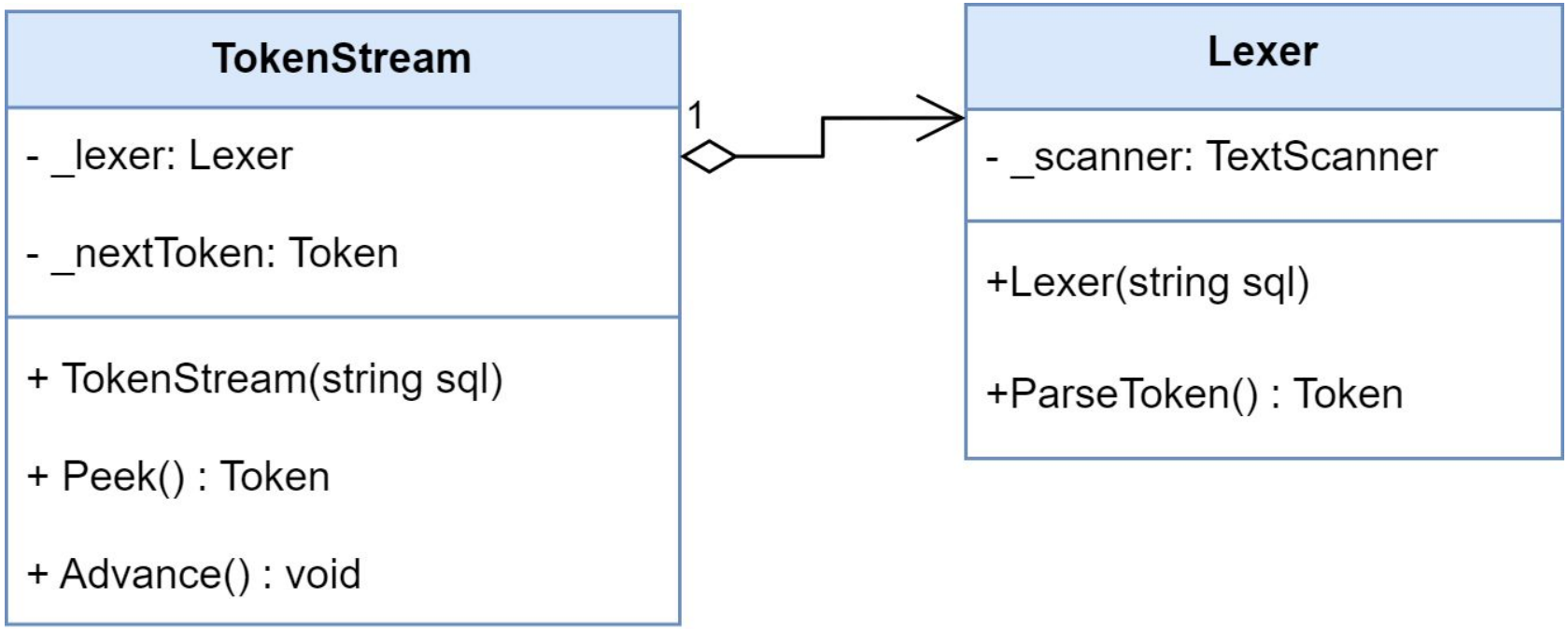
```
factor = identifier
    | identifier, "(", expression_list, ")"
    | number
```

Рекурсивный спуск

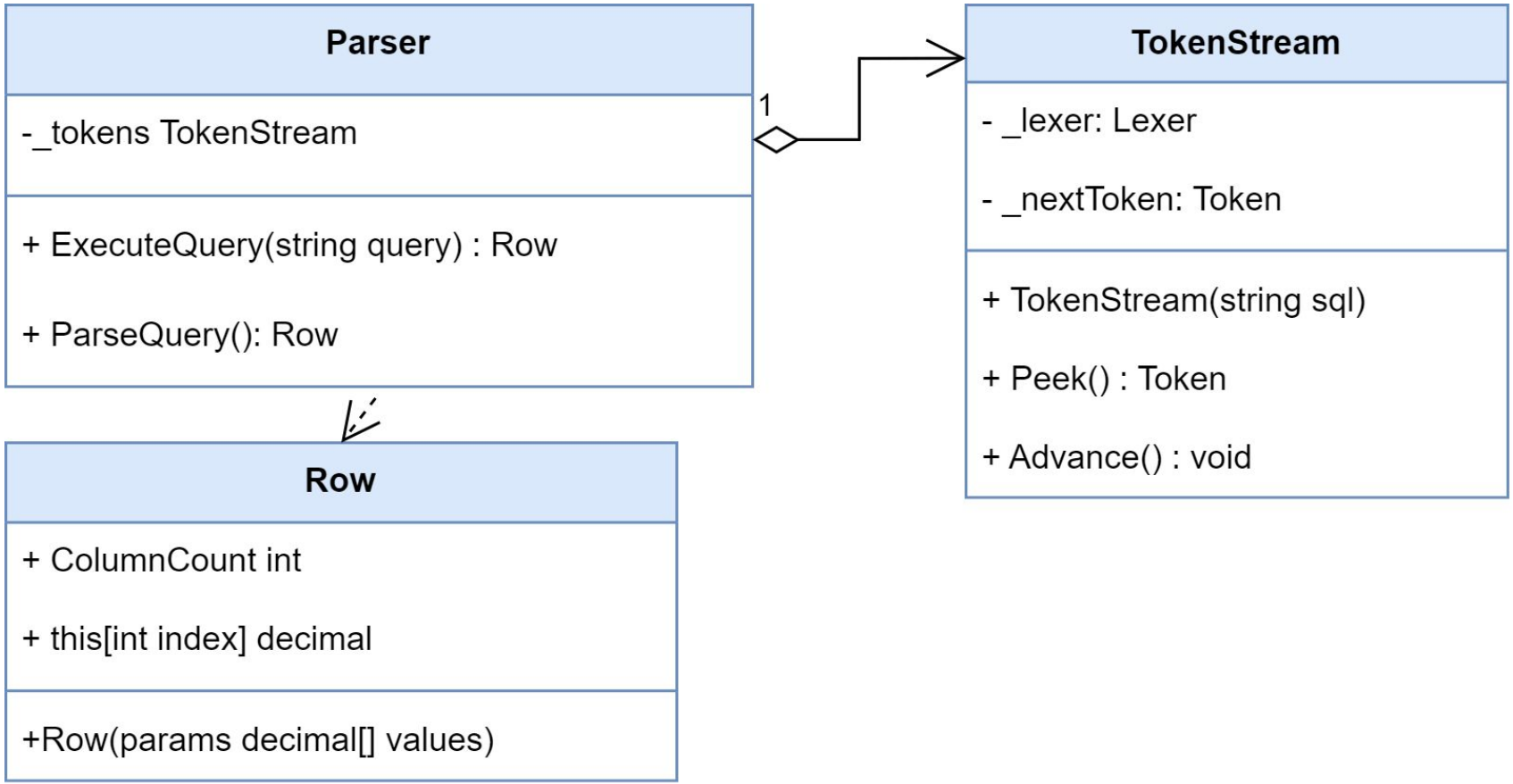
Список тестов

- [ ] Разбор SELECT без FROM
- [ ] Разбор SELECT без завершающего разделителя
- [ ] Разбор сложения и вычитания
- [ ] Разбор списка выражений
- [ ] Учёт левой ассоциативности при вычитании
- [ ] Разбор умножения и деления
- [ ] Разбор с разными приоритетами операторов
- [ ] Разбор скобок
- [ ] Разбор вызова функций
- ...

Предпросмотр на один токен



Класс парсера



Класс парсера

```
public class Parser(string sql) {
    private readonly TokenStream _tokens = new(sql);

    // Выполняет SQL-запрос и возвращает результат.
    public static Row ExecuteQuery(string query) {
        Parser p = new(query);
        return p.ParseQuery();
    }
}
```

Превращаем правила в методы

```
/// <summary>
///  Разбирает одно выражение.
///  Правила:
///   expression = term_expression,
//         { ("+" | "-"), term_expression } ;
/// </summary>
private decimal ParseExpression()
```

Класс парсера

```
// Выполняет SQL-запрос и возвращает результат.
// Поддерживает правила:
//  query = select_statement, [ ";" ] ;
private Row ParseQuery() {
    Row result = ParseSelectQuery();
    ParseQueryDelimiter();
    return result;
}
```

Класс парсера

```
// Разбирает SELECT-запрос без FROM.
// Правила:
//     select_statement = "SELECT", expression_list ;
private Row ParseSelectQuery() {
    SkipExpectedLexeme(TokenType.Select);
    List<decimal> values = ParseExpressionList();
    return new Row(values.ToArray());
}
```

Левоассоциативные операции — вариант №1

```
private decimal ParseExpression() {
    decimal value = ParseTermExpression();
    switch (_tokens.Peek().Type) {
        case TokenType.PlusSign:
            _tokens.Advance();
            return value + ParseExpression();
        case TokenType.MinusSign: // аналогично
        default:
            return value;
    }
```

Левоассоциативные операции — вариант №1

```
private decimal ParseExpression() {
    decimal value = ParseTermExpression();
    switch (_tokens.Peek().Type) {
        case TokenType.PlusSign:
            _tokens.Advance();
            return value + ParseExpression();
        case TokenType.MinusSign: // аналогично
        default:
            return value;
    }
```

Неверно обработает
3 - 8 + 7

Левоассоциативные операции — вариант №2

```
while (true) {
    switch (_tokens.Peek().Type)  {
        case TokenType.PlusSign:
            _tokens.Advance();
            value += ParseTermExpression();
            break;
        case TokenType.MinusSign: // аналогично
        default:
            return value;
}
```

Верно обработает
3 - 8 + 7

Вызовы функций

```
// function_call = function_name, "(", [ args ], ")" ;
private decimal ParseFunctionCall() {
    string name = _tokens.Peek().Value!.ToString();
    Match(TokenType.Identifier);
    Match(TokenType.OpenParenthesis);
    List<decimal> arguments = ParseExpressionList();
    Match(TokenType.CloseParenthesis);
    return BuiltinFunctions.Invoke(name, arguments);
}
```

Вызовы функций

```
// function_call = function_name, "(", [ args ], ")" ;

private decimal ParseFunctionCall() {
    string name = _tokens.Peek().Value!.ToString();
    Match(TokenType.Identifier);
    Match(TokenType.OpenParenthesis);
    List<decimal> arguments = ParseExpressionList();
    Match(TokenType.CloseParenthesis);
    return BuiltinFunctions.Invoke(name, arguments);
}
```

Вызовы функций

```
// function_call = function_name, "(", [ args ], ")" ;

private decimal ParseFunctionCall() {
    string name = _tokens.Peek().Value!.ToString();
    Match(TokenType.Identifier);
    Match(TokenType.OpenParenthesis);
    List<decimal> arguments = ParseExpressionList();
    Match(TokenType.CloseParenthesis);
    return BuiltinFunctions.Invoke(name, arguments);
}
```

Вызовы функций

```
// function_call = function_name, "(", [ args ], ")" ;

private decimal ParseFunctionCall() {
    string name = _tokens.Peek().Value!.ToString();
    Match(TokenType.Identifier);
    Match(TokenType.OpenParenthesis);
    List<decimal> arguments = ParseExpressionList();
    Match(TokenType.CloseParenthesis);
    return BuiltinFunctions.Invoke(name, arguments);
}
```

Вызовы функций

```
// function_call = function_name, "(", [ args ], ")" ;
private decimal ParseFunctionCall() {
    string name = _tokens.Peek().Value!.ToString();
    Match(TokenType.Identifier);
    Match(TokenType.OpenParenthesis);
    List<decimal> arguments = ParseExpressionList();
    Match(TokenType.CloseParenthesis);
    return BuiltinFunctions.Invoke(name, arguments);
}
```

Вызовы функций

```
// function_call = function_name, "(", [ args ], ")" ;
private decimal ParseFunctionCall() {
    string name = _tokens.Peek().Value!.ToString();
    Match(TokenType.Identifier);
    Match(TokenType.OpenParenthesis);
    List<decimal> arguments = ParseExpressionList();
    Match(TokenType.CloseParenthesis);
    return BuiltinFunctions.Invoke(name, arguments);
}
```

Операция Match

```
// Пропускает ожидаемую лексему либо бросает исключение,
// если встретит иную лексему.
private void Match(TokenType expected) {
    Token t = _tokens.Peek();
    if (t.Type != expected) {
        throw new UnexpectedLexemeException(expected, t);
    }
    _tokens.Advance();
}
```

Исходный код примера

https://sourcecraft.dev/sshambir-public/memsql

- `docs/specification/` — описание языка
- `src/SqlParser` — исходный код
- `tests/SqlParser.UnitTests` — тесты

Конец!
Вопросы?