

Національний технічний університет України  
«Київський політехнічний інститут імені Ігоря Сікорського»

Факультет інформатики та обчислювальної техніки  
Кафедра обчислювальної техніки

## **ЗВІТ**

**з лабораторної роботи №1**  
**з дисципліни ”Програмне забезпечення комп’ютерних систем”**

**Тема: Лексичний та синтаксичний аналіз**

**Варіант №5**

Виконав:  
Студент 1 курсу, групи ІМ-51мн  
Ковальов Олександр

Перевірила:  
к.т.н., Русанова Ольга Веніаминівна

Дата здачі: 14.10.2025

**Мета роботи:** навчитися виконувати лексичний та синтаксичний аналіз заданого арифметичного виразу.

**Вхідні дані:** арифметичний вираз з дужками або в бездужковому записі, елементами якого є константи, імена змінних, алгебраїчні операції та математичні функції.

**Завдання:** реалізувати лексичний та синтаксичний аналізатор арифметичного виразу з використанням будь-якої мови програмування. Необхідно, щоб аналізатор перевіряв такі типи помилок:

1. Помилки на початку арифметичного виразу (наприклад, вираз не може починатись із закритої дужки, алгебраїчних операцій \* та /);
2. Помилки, пов'язані з неправильним написанням імен змінних, констант та при необхідності функцій;
3. Помилки у кінці виразу (наприклад, вираз не може закінчуватись будь-якою алгебраїчною операцією);
4. Помилки в середині виразу (подвійні операції, відсутність операцій перед або між дужками, операції \* або / після відкритої дужки тощо);
5. Помилки, пов'язані з використанням дужок (нерівна кількість відкритих та закритих дужок, неправильний порядок дужок, пусті дужки).

Синтаксичний аналізатор потрібно реалізувати за допомогою кінцевого автомату. Результатом виконання даної лабораторної роботи є список помилок, виявлених під час синтаксичного аналізу.

### Хід роботи.

В ході лабораторної роботи був написаний застосунок мовою програмування Rust. Він складається з декількох частин – візуальна частина (фронтенд), яка надає консольний інтерфейс користувачу. Обчислення відбуваються на частині, яка відповідає за логіку (бекенд). Вона поділяється на модулі, які відповідають за лексичний (токенізація) та синтаксичний аналіз.

```
xairaven@laptop ~/Files/Programming/KPI-SCS <main>
$ cargo run --bin Lab1 -- --help
   Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.01s
    Running `target/debug/Lab1 --help`
Usage: Lab1 [OPTIONS] --code-file <CODE_FILE>

Options:
  -c, --code-file <CODE_FILE>    Code file.
  -o, --output-file <OUTPUT_FILE> Output file name. If not provided, output will be printed to console.
  -p, --pretty                    Pretty print output.
  -h, --help                      Print help
  -V, --version                  Print version
xairaven@laptop ~/Files/Programming/KPI-SCS <main>
$
```

Лексичний аналіз (токенізація) потрібен для формалізації вхідного потоку даних. Тобто, кожен логічний частинку тексту можна виділити для того, щоб мати змогу в подальшому їх застосовувати набагато ефективніше, особливо враховуючи систему типів обраної мови програмування. За допомогою типу даних "enum" та додаткової логіки можна повністю класифікувати весь код.

Були виділені основні типи токенів: ідентифікатор, число, знаки плюс, мінус, зірочка, слеш, відсоток, круглі дужки, квадратні дужки, знак оклику, амперсанд, пряма лінія, крапка, кома, лапки, пробіл, таб, нова лінія, та насамкінець невідомий тип токена.

Токен типу "Ідентифікатор" включає в себе всі послідовні символи, якщо вони починаються з букви та містять в собі цифри та нижнє підчеркування. Числа складаються з цифр від 0 до 9. Пробіл містить в собі всі послідовні пробіли. Всі інші типи токенів складаються з відповідних та інтуїтивних символів, по одному на токен.

Сама структура "Токен" складається з трьох полів: тип, позиція в тексті (діапазон) та опціональне значення (потрібне для тексту, чисел, невідомих символів).

Синтаксичний аналіз більш складний. Базова реалізація полягає в абстракції кінцевого автомата. По-перше, для аналізу потрібен контекст - ним виступає окрема структура, яка містить вектор токенів, поточний індекс (для ітерації по токенах), структура "Статус" (для зберігання стану в логічних змінних), вектор синтаксичних помилок, та стеки круглих, квадратних дужок і лапок.

Типів синтаксичних помилок всього 20: "порожні квадратні дужки", "порожні круглі дужки", "некоректний бінарний літерал", "некоректний шістнадцятковий літерал", неправильне число з плаваючою комою, неправильна назва функції, некоректна назва змінної, неочікувані квадратні дужки, коми, крапки, кінець виразу, нова лінія, операнд, оператор, круглі дужки. Окремо є помилки "невідомий токен", непарні квадратні дужки, круглі дужки, лапки.

Структура "Статус" має три поля: "очікуємо операнд", "очікуємо оператор", та "знаходимося в рядку".

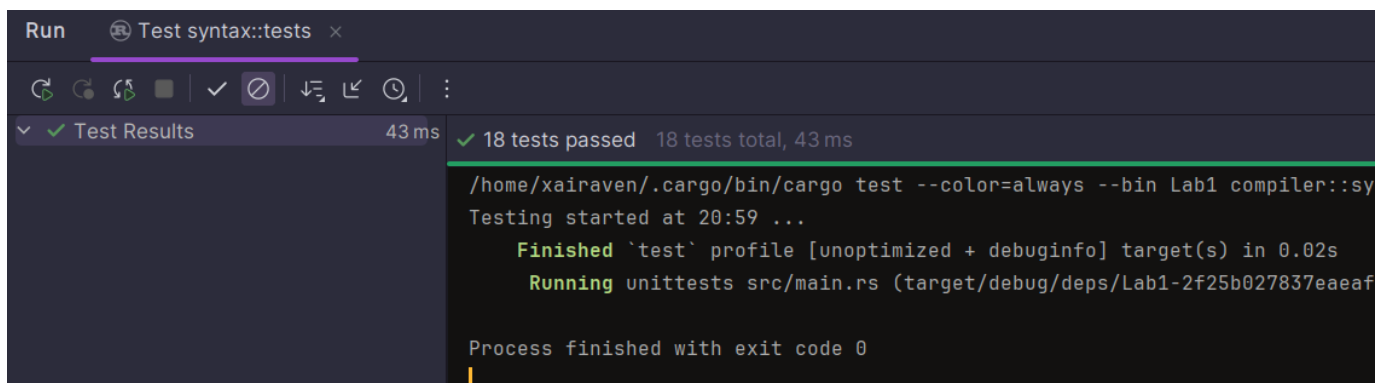
На початку аналізу встановлюється статус "Очікуємо операнд". Далі, з вектору токенів видаляються всі пробіли та таби, які не належать певному рядку.

Після попередньої підготовки вхідного потоку (видалення непотрібних пробілів та табуляцій поза рядковими літералами) виконується поетапний лінійний прохід по вектору токенів. На кожному кроці кінцевий автомат

розглядає поточний токен та контекст (поля структури `Status`, стеки для дужок і лапок, попередні та наступні токени) і приймає одне з наступних рішень:

1. Зрушити індекс на один (типовий випадок для одиничних токенів: оператори, роздільники, прості літерали);
2. Зрушити індекс на кілька позицій (складні літерали: число-точка-число для чисел з плаваючою крапкою, або префіксні системи числення, які токенізуються як два токени — наприклад `0` та `xFF`);
3. Записати синтаксичну помилку в масив `errors` з вказанням токена та типу помилки;
4. Змінити внутрішній стан автомата (очікуємо операнд / очікуємо оператор / всередині рядка) та оновити стеки дужок/лапок.

**Тестування і приклади результатів.** Для перевірки коректності реалізації створено набір автоматичних юніт-тестів (функції з префіксом `test_syntax_*`), які моделюють різні види помилок: початкові помилки (наприклад, вираз починається з оператора), подвійні оператори, невірні імена змінних та функцій, неправильні числові форми, незбалансовані дужки та незакриті лапки. Загалом реалізовано 18 тестів, що охоплюють як прості, так і складні синтаксичні конструкції.



The screenshot shows a terminal window with the title 'Run' and a sub-title 'Test syntax::tests'. The terminal output displays the results of running tests: '18 tests passed 18 tests total, 43 ms'. Below this, it shows the command used: `/home/xairaven/.cargo/bin/cargo test --color=always --bin Lab1 compiler::sy`. The output continues with 'Testing started at 20:59 ...', 'Finished `test` profile [unoptimized + debuginfo] target(s) in 0.02s', 'Running unittests src/main.rs (target/debug/deps/Lab1-2f25b027837eaeaf)', and 'Process finished with exit code 0'.

Запуск тестів у середовищі розробки Rust виконується стандартною командою:

```
1 cargo test --package Lab1 --bin Lab1
2
```

Приклад виводу помилок для виразу "-a ++ b - 2v\*func((t+2 -, sin(x/\*2.01.2), )/8(-)\*\*" (перший юніт-тест):

1	-a ++ b - 2v*func((t+2 -, sin(x/*2.01.2), )/8(-)**	
2	^ ^ ^ ^ ^ ^ ^ ^ ^	
3		Unexpected end of expression. [Position: 50]
4		Unexpected operator. [Position: 50]
5		Unexpected parenthesis. [Position: 48]
6		Unexpected operator. [Position: 47]
7		Unexpected function name '8'. [Position: 45]
8		Missing function argument. [Position: 41]
9		Unexpected operand '2'. [Position: 39]
10		Unexpected dot. [Position: 38]
11		Unexpected operator. [Position: 33]
12		Unexpected comma. [Position: 25]
13		Unmatched parenthesis. [Position: 18]
14		Invalid variable name. [Position: 11]
15		Unexpected operator. [Position: 5]
16		

Формат повідомлення формується методом `SyntaxError::display(column_length)` – помилка (з типом та додочною інформацією, якщо є) + позиція у виразі. Це дозволяє користувачу швидко локалізувати і виправити синтаксичну проблему.

**Аналіз складності.** Алгоритм реалізовано як єдиний лінійний прохід по списку токенів з підтримкою стеків для дужок і лапок. Тому часові витрати оцінюються як  $O(n)$ , де  $n$  – кількість токенів. Пам'яткова складність також лінійна  $O(n)$  (вектор токенів + стекові структури для вкладених дужок / лапок). Додаткові валідації літералів (перевірка hex / binary, перевірка float) виконуються за час, пропорційний довжині відповідного токена, що не змінює асимптотики.

**Обмеження поточної реалізації.** У роботі зроблено низку свідомих спрощень, які варто враховувати:

- Аналізатор зосереджений на виявленні синтаксичних помилок; він не буде повного AST і не виконує семантичну перевірку (типізацію, перевірку сигнатур функцій, області видимості тощо).
- Обмежена модель імен: рядок правила для ідентифікаторів та функцій є спрощеним (перевірка початку з букви та дозволених символів), тому в деяких граничних випадках реальні мови з іншими правилами імен можуть продукувати некоректні попередження.
- Валідація чисел (особливо складних варіантів: експоненційна форма, локальні формати) реалізована частково.
- Механізму відновлення після помилок (error recovery) поки що немає —

після виявлення помилки алгоритм продовжує з поточної позиції, що іноді призводить до лавинних помилок у подальшій частині виразу.

**Практичне застосування та інтерфейс.** Аналізатор легко інтегрувати в іншу консольну утиліту чи IDE-плагін: достатньо викликати модуль токенізації над рядком коду, передати вектор токенів у `SyntaxAnalyzer::new(tokens)` та отримати назад вектор помилок через `analyze()`. У прикладній консолі можна виводити помилки з підсвіткою відповідної позиції у тексті або формувати звіт у текстовому/JSON-форматі для подальшої обробки.

#### **Пропозиції для подальшого розширення (детальніше):**

1. **Побудова AST:** після синтаксичної перевірки додати етап побудови абстрактного синтаксичного дерева – це відкриє шлях до семантичної перевірки, оптимізації та виконання виразів.
2. **Покращене розпізнавання літералів:** реалізувати повністю стандартні форми чисел (наприклад, експоненційну записку  $1.23e-4$ ), а також підтримку підкреслень у числових літералах для кращої читабельності (як у сучасних мовах).
3. **Міжнародні та локалізовані формати:** зробити парсер більш гнучким щодо роздільників десяткових частин (кома/крапка) – з переключенням за локаллю.
4. **Механізми відновлення після помилок:** застосувати стратегії «panic mode» або «phrase-level recovery», щоб знаходити кілька незалежних помилок у одному проході без каскаду помилок.
5. **Плагін до редактора/IDE:** інтегрувати модуль як LSP-сервер або розширення для VSCode/IntelliJ, щоб користувачі отримували підказки та підсвітку помилок в режимі реального часу.
6. **Розширені тести:** автоматизувати побудову інфраструктури для property-based тестів (наприклад, за допомогою quickcheck), що дозволить генерувати випадкові вирази та перевіряти стабільність аналізатора.

**Висновок.** В ході виконання лабораторної роботи було розроблено та реалізовано лексичний і синтаксичний аналізатор арифметичних виразів мовою Rust. Лексична частина реалізує виділення ключових типів токенів та формування структури Token з позицією та опціональним значенням, що суттєво спрощує наступні етапи аналізу. Синтаксичний аналізатор побудовано у вигляді кінцевого автомата з явними станами (очікуємо операнд, очікуємо

оператор, всередині рядка) та стековими структурами для відстеження вкладених дужок і лапок. Реалізовано набір із двадцяти типів синтаксичних помилок, що охоплює найпоширеніші некоректні конструкції (помилки позиційної структури, некоректні літерали, незбалансовані дужки, порожні аргументи тощо).

Практичні результати показали: реалізація коректно виявляє помилки та повертає інформативні повідомлення з точною позицією у виразі; набір юніт-тестів демонструє відповідність очікуваним результатам у широкому спектрі помилкових та граничних прикладів. З точки зору продуктивності рішення є ефективним – лінійний часовий прохід та лінійні пам'яткові витрати дозволяють застосовувати аналізатор для рядків довільної довжини у інструментальних сценаріях.

На завершення варто підкреслити навчальну цінність роботи: реалізація надала практичний досвід проектування токенайзера, кінцевого автомата для синтаксичного аналізу, написання юніт-тестів та формування зручних повідомлень про помилки. Пропоновані напрямки подальшої роботи (побудова AST, поліпшення валідації літералів, механізми відновлення після помилок, інтеграція в IDE) дозволять перетворити цей аналізатор на більш універсальний та промисловоздатний інструмент для обробки математичних та програмних виразів.

## Програмний код.

### tokenizer.rs:

```
1 use std::ops::Range;
2 use strum_macros::Display;
3
4 #[derive(Debug, Clone, PartialEq, Eq)]
5 pub struct Token {
6     pub kind: TokenType,
7     pub position: Range,
8     pub value: Option<String>,
9 }
10
11 impl Token {
12     pub fn display_position(&self) -> String {
13         if self.position.start + 1 == self.position.end {
14             format!("[Position: {}]", self.position.start + 1)
15         } else {
16             format!(
17                 "[Position: {}..{}]",
18                 self.position.start + 1,
19                 self.position.end
```

```

20     )
21     }
22 }
23 }
24
25 #[derive(Debug, Clone, PartialEq, Eq, Display)]
26 pub enum TokenType {
27     Identifier,
28     Number,
29
30     Plus,
31     Minus,
32     Asterisk,
33     Slash,
34     Percent,
35
36     LeftParenthesis,
37     RightParenthesis,
38     LeftBracket,
39     RightBracket,
40
41     ExclamationMark,
42     Ampersand,
43     Pipe,
44
45     Dot,
46     Comma,
47
48     QuotationMark,
49
50     Space,
51     Tab,
52     NewLine,
53
54     Unknown,
55 }
56
57 macro_rules! token {
58     ($token_type:expr, $position:literal) => {
59         Token {
60             kind: $token_type,
61             position: $position..($position + 1),
62             value: None,
63         }
64     };
65     ($token_type:expr, $position:expr) => {
66         Token {
67             kind: $token_type,
68             position: $position,
69             value: None,

```



```

70     }
71 };
72 ($token_type:expr, $value:expr, $position:literal) => {
73     Token {
74         kind: $token_type,
75         position: $position..($position + 1),
76         value: Some($value),
77     }
78 };
79 ($token_type:expr, $value:expr, $position:expr) => {
80     Token {
81         kind: $token_type,
82         position: $position,
83         value: Some($value),
84     }
85 };
86 }
87
88 pub fn tokenize(input: &str) -> Vec<Token> {
89     let mut tokens: Vec<Token> = Vec::new();
90     let chars: Vec<char> = input.chars().collect();
91
92     for (index, symbol) in chars.iter().enumerate() {
93         if let Some(last_token) = tokens.last()
94         && last_token.position.end > index
95         {
96             continue;
97         }
98
99         let token = match symbol {
100             symbol if symbol.is_alphabetic() || symbol.eq(&'_') => {
101                 let start = index;
102                 let mut end = index + 1;
103
104                 while end < chars.len()
105                 && (chars[end].is_alphanumeric() || chars[end] == '_')
106                 {
107                     end += 1;
108                 }
109
110                 let value: String = chars[start..end].iter().collect();
111                 token!(TokenType::Identifier, value, start..end)
112             },
113             '0'..'9' => {
114                 let start = index;
115                 let mut end = index + 1;
116
117                 while end < chars.len() && chars[end].is_numeric() {
118                     end += 1;
119                 }

```

```

120
121     let value: String = chars[start..end].iter().collect();
122     token!(TokenType::Number, value, start..end)
123 },
124 '+' => token!(TokenType::Plus, index..index + 1),
125 '-' => token!(TokenType::Minus, index..index + 1),
126 '*' => token!(TokenType::Asterisk, index..index + 1),
127 '/' => token!(TokenType::Slash, index..index + 1),
128 '%' => token!(TokenType::Percent, index..index + 1),
129 '(' => token!(TokenType::LeftParenthesis, index..index + 1),
130 ')' => token!(TokenType::RightParenthesis, index..index + 1),
131 '[' => token!(TokenType::LeftBracket, index..index + 1),
132 ']' => token!(TokenType::RightBracket, index..index + 1),
133 '!' => token!(TokenType::ExclamationMark, index..index + 1),
134 '&' => token!(TokenType::Ampersand, index..index + 1),
135 '|' => token!(TokenType::Pipe, index..index + 1),
136 '.' => token!(TokenType::Dot, index..index + 1),
137 ',' => token!(TokenType::Comma, index..index + 1),
138 '"' => token!(TokenType::QuotationMark, index..index + 1),
139 '\n' => token!(TokenType::NewLine, index..index + 1),
140 c if c.eq('&\'t') => token!(TokenType::Tab, index..index + 1),
141 c if c.is_whitespace() => {
142     let start = index;
143     let mut end = index + 1;
144
145     while end < chars.len() && chars[end].is_whitespace() {
146         end += 1;
147     }
148
149     token!(TokenType::Space, start..end)
150 },
151 c => token!(TokenType::Unknown, c.to_string(), index..index + 1),
152 };
153
154 tokens.push(token);
155 }
156
157 tokens
158 }
159

```

## syntax.rs:

```

1 use crate::compiler::tokenizer::{Token, TokenType};
2 use colored::Colorize;
3 use std::collections::VecDeque;
4
5 #[derive(Debug)]
6 pub struct SyntaxAnalyzer {
7     tokens: Vec<Token>,
8     current_index: usize,

```

```

9
10     status: Status,
11     errors: Vec<SyntaxError>,
12
13     brackets_stack: VecDeque<Token>,
14     parentheses_stack: VecDeque<Token>,
15     quotation_marks_stack: VecDeque<Token>,
16 }
17
18 #[derive(Debug, PartialEq, Eq)]
19 pub struct SyntaxError {
20     pub token: Token,
21     pub kind: SyntaxErrorKind,
22 }
23
24 macro_rules! syntax_error {
25     ($kind:ident, $token:expr) => {
26         SyntaxError {
27             token: $token.clone(),
28             kind: SyntaxErrorKind::$kind,
29         }
30     };
31 }
32
33 #[derive(Debug, PartialEq, Eq)]
34 pub enum SyntaxErrorKind {
35     EmptyBrackets,
36     EmptyParentheses,
37     InvalidBinaryLiteral,
38     InvalidFloat,
39     InvalidFunctionName,
40     InvalidHexLiteral,
41     InvalidVariableName,
42     MissingArgument,
43     UnexpectedBrackets,
44     UnexpectedComma,
45     UnexpectedDot,
46     UnexpectedEndOfExpression,
47     UnexpectedNewLine,
48     UnexpectedOperand,
49     UnexpectedOperator,
50     UnexpectedParenthesis,
51     UnknownToken,
52     UnmatchedBrackets,
53     UnmatchedParenthesis,
54     UnmatchedQuotationMark,
55 }
56
57 impl std::fmt::Display for SyntaxError {
58     fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {

```

```

59     let text = match self.kind {
60         SyntaxErrorKind::EmptyBrackets => "Empty array access.",
61         SyntaxErrorKind::EmptyParentheses => "Empty function or grouping.",
62         SyntaxErrorKind::InvalidBinaryLiteral => match &self.token.value {
63             None => "Invalid binary literal.",
64             Some(value) => &format!("Invalid binary literal '{0}'.", value),
65         },
66         SyntaxErrorKind::InvalidFloat => "Invalid float.",
67         SyntaxErrorKind::InvalidFunctionName => match &self.token.value {
68             None => "Unexpected function name.",
69             Some(value) => &format!("Unexpected function name '{0}'.", value),
70         },
71         SyntaxErrorKind::InvalidHexLiteral => match &self.token.value {
72             None => "Invalid hexadecimal literal.",
73             Some(value) => &format!("Invalid hexadecimal literal '{0}'.", value),
74         },
75         SyntaxErrorKind::InvalidVariableName => "Invalid variable name.",
76         SyntaxErrorKind::MissingArgument => "Missing function argument.",
77         SyntaxErrorKind::UnexpectedBrackets => "Unexpected brackets.",
78         SyntaxErrorKind::UnexpectedComma => "Unexpected comma.",
79         SyntaxErrorKind::UnexpectedDot => "Unexpected dot.",
80         SyntaxErrorKind::UnexpectedEndOfExpression => "Unexpected end of expression.",
81         SyntaxErrorKind::UnexpectedNewLine => "Unexpected newline.",
82         SyntaxErrorKind::UnexpectedOperand => match &self.token.value {
83             None => "Unexpected operand.",
84             Some(value) => &format!("Unexpected operand '{0}'.", value),
85         },
86         SyntaxErrorKind::UnexpectedOperator => "Unexpected operator.",
87         SyntaxErrorKind::UnexpectedParenthesis => "Unexpected parenthesis.",
88         SyntaxErrorKind::UnknownToken => "Unknown token.",
89         SyntaxErrorKind::UnmatchedBrackets => "Unmatched brackets.",
90         SyntaxErrorKind::UnmatchedParenthesis => "Unmatched parenthesis.",
91         SyntaxErrorKind::UnmatchedQuotationMark => "Unmatched quotation mark.",
92     };
93
94     write!(f, "{}", text)
95 }
96
97
98 impl SyntaxError {
99     pub fn display(&self, column_length: usize) -> String {
100         format!(
101             "{:fill$} {}",
102             self.to_string().bold().red(),
103             self.token.display_position().bold(),
104             fill = column_length,
105         )
106     }
107 }
108

```

```

109 #[derive(Debug, Default)]
110 pub struct Status {
111     pub expect_operand: bool,
112     pub expect_operator: bool,
113     pub in_string: bool,
114 }
115
116 impl SyntaxAnalyzer {
117     pub fn new(tokens: Vec<Token>) -> Self {
118         Self {
119             tokens,
120             current_index: 0,
121
122             errors: Vec::new(),
123             status: Status::default(),
124
125             brackets_stack: VecDeque::new(),
126             parentheses_stack: VecDeque::new(),
127             quotation_marks_stack: VecDeque::new(),
128         }
129     }
130
131     pub fn analyze(mut self) -> Vec<SyntaxError> {
132         self.status = Status {
133             expect_operand: true,
134             expect_operator: false,
135             in_string: false,
136         };
137
138         // Deleting redundant spaces & tabs
139         {
140             let mut delete_spaces = Vec::new();
141             let mut in_string = false;
142             for (i, token) in self.tokens.iter().enumerate() {
143                 if token.kind == TokenType::QuotationMark {
144                     in_string = !in_string;
145                 }
146                 if !in_string
147                     && (token.kind == TokenType::Space || token.kind == TokenType::Tab)
148                 {
149                     delete_spaces.push(i);
150                 }
151             }
152             for index in delete_spaces.iter().rev() {
153                 self.tokens.remove(*index);
154             }
155         }
156
157         while self.current_index < self.tokens.len() {
158             let token = &self.tokens[self.current_index];

```

```

159
160     match &token.kind {
161         TokenType::QuotationMark => {
162             // Toggle string state.
163             if !self.status.in_string {
164                 // Start mark. We're expecting an operand here.
165                 if !self.status.expect_operand {
166                     // If we didn't expect an operand, it's an error.
167                     self.errors.push(syntax_error!(UnexpectedOperator, token));
168                 }
169                 self.status.in_string = true;
170                 // While inside string we're considering that operand is not finished
171                 self.status.expect_operator = false;
172             } else {
173                 // Closing mark
174                 self.status.in_string = false;
175                 // String literal is operand
176                 self.status.expect_operator = true;
177             }
178
179             if self.quotation_marks_stack.is_empty() {
180                 self.quotation_marks_stack.push_back(token.clone());
181             } else {
182                 self.quotation_marks_stack.pop_back();
183             }
184
185             self.status.expect_operand = false;
186             self.current_index += 1;
187             continue;
188         },
189
190         _ if self.status.in_string => {
191             self.current_index += 1;
192             continue;
193         },
194
195         TokenType::ExclamationMark => {
196             // Used only like identifier part
197             if self.status.expect_operand {
198                 self.status.expect_operand = true;
199                 self.status.expect_operator = false;
200             } else {
201                 self.errors.push(syntax_error!(UnexpectedOperator, token));
202                 // Continuing, but considering that operator was read.
203             }
204             self.current_index += 1;
205             continue;
206         },
207
208         TokenType::Identifier => {

```

```

209 // Identifier - operand
210 if !self.status.expect_operand {
211     self.errors.push(syntax_error!(UnexpectedOperand, token));
212     // Continuing, but considering that operand was read
213 }
214 self.status.expect_operand = false;
215 self.status.expect_operator = true;
216 self.current_index += 1;
217 continue;
218 },
219
220 TokenType::Number => {
221     // Number - operand
222     if !self.status.expect_operand {
223         self.errors.push(syntax_error!(UnexpectedOperand, token));
224         self.current_index += 1;
225         continue;
226     }
227
228     // Binary and Hex validating
229     if let Some(prefix) = &token.value
230     && prefix.eq("0")
231     && let Some(next) = self.peek_next()
232     && next.kind == TokenType::Identifier
233     && let Some(value) = &next.value
234     && value.to_ascii_lowercase().starts_with(['x', 'b'])
235     && value.len() > 1
236     {
237         // Hex
238         if value.to_ascii_lowercase().starts_with('x')
239         && !value[1..].chars().all(|c| c.is_ascii_hexdigit())
240         {
241             // Incorrect hex literal
242             self.errors.push(syntax_error!(InvalidHexLiteral, next));
243         }
244         // Binary
245         else if value.to_ascii_lowercase().starts_with('b')
246         && !value[1..].chars().all(|c| c == '0' || c == '1')
247         {
248             // Incorrect binary literal
249             self.errors.push(syntax_error!(InvalidBinaryLiteral, next));
250         }
251
252         // Anyway, considering that identifier was read
253         self.current_index += 2;
254         self.status.expect_operand = false;
255         self.status.expect_operator = true;
256         continue;
257     }
258

```

```

259 // Float validating
260 if let Some(next) = self.peek_next()
261 && next.kind == TokenType::Dot
262 {
263     if let Some(second) = self.peek_next_by(2) {
264         if matches!(&second.kind, TokenType::Number) {
265             // Correct float! Number-Dot-Number
266             // Next token - the third
267             self.current_index += 3;
268         } else {
269             // Something else after dot - error
270             self.errors.push(syntax_error!(InvalidFloat, next));
271             // Skipping number with the dot
272             self.current_index += 2;
273         }
274     } else {
275         // Dot in the end - error
276         self.errors.push(syntax_error!(UnexpectedOperator, next));
277         self.current_index += 2;
278     }
279     self.status.expect_operand = false;
280     self.status.expect_operator = true;
281     continue;
282 }
283
284 // Bad variable name?
285 if let Some(next) = self.peek_next()
286 && next.kind == TokenType::Identifier
287 {
288     // But if second next identifier is left parentheses - it's function name
289     if let Some(second) = self.peek_next_by(2)
290     && second.kind == TokenType::LeftParenthesis
291     {
292         // Function name cannot start with a number
293         self.errors.push(syntax_error!(InvalidFunctionName, token));
294     } else {
295         // If next token is identifier, then it's bad variable name
296         self.errors.push(syntax_error!(InvalidVariableName, token));
297     }
298
299     // Skipping invalid identifier
300     self.current_index += 2;
301     self.status.expect_operand = false;
302     self.status.expect_operator = true;
303     continue;
304 }
305
306 // Integer literal
307 self.current_index += 1;
308 self.status.expect_operand = false;

```



```

309     self.status.expect_operator = true;
310     continue;
311 },
312
313 TokenType::Dot => {
314     self.errors.push(syntax_error!(UnexpectedDot, token));
315     self.current_index += 1;
316     continue;
317 },
318
319 // Mathematical and logical operations
320 TokenType::Plus
321 | TokenType::Minus
322 | TokenType::Asterisk
323 | TokenType::Slash
324 | TokenType::Percent
325 | TokenType::Ampersand
326 | TokenType::Pipe => {
327     // Unary operations
328     let unary = if [TokenType::Minus].contains(&token.kind)
329     && let Some(next) = self.peek_next()
330     && [
331         TokenType::Identifier,
332         TokenType::Number,
333         TokenType::LeftParenthesis,
334     ]
335     .contains(&next.kind)
336     {
337         true
338     } else {
339         false
340     };
341
342     if self.status.expect_operator || unary {
343         self.status.expect_operand = true;
344         self.status.expect_operator = false;
345     } else {
346         self.errors.push(syntax_error!(UnexpectedOperator, token));
347         // Waiting for operand still
348     }
349     self.current_index += 1;
350     continue;
351 },
352
353 TokenType::LeftBracket => {
354     // LeftBracket can be there if previous token is Identifier (array access)
355     // or that's array with more than one dimension (e.g. arr[2][3])
356     let allow = matches!(self.peek_previous(), Some(t) if matches!(t.kind, TokenType::
Identifier))
357     || matches!(self.peek_previous(), Some(t) if matches!(t.kind, TokenType::RightBracket));

```

```

358     if !allow {
359         self.errors.push(syntax_error!(UnexpectedBrackets, token));
360         self.current_index += 1;
361         continue;
362     }
363
364     self.brackets_stack.push_back(token.clone());
365     self.status.expect_operand = true;
366     self.status.expect_operator = false;
367     self.current_index += 1;
368     continue;
369 },
370
371 TokenType::RightBracket => {
372     match self.brackets_stack.pop_back().is_some() {
373         true => {
374             // Correct
375             self.status.expect_operand = false;
376             self.status.expect_operator = true;
377         },
378         false => {
379             self.errors.push(syntax_error!(UnmatchedBrackets, token))
380         },
381     }
382
383     // Empty array access check
384     if let Some(previous) = self.peek_previous()
385     && matches!(previous.kind, TokenType::LeftBracket)
386     {
387         self.errors.push(syntax_error!(EmptyBrackets, token));
388     }
389
390     self.current_index += 1;
391     continue;
392 },
393
394 TokenType::LeftParenthesis => {
395     // LeftParenthesis can be there if we're waiting for operand (grouping)
396     // or previous token is Identifier (function call)
397     // Number - error (processing later)
398     // RightParenthesis - error (processing later)
399     let allow = self.status.expect_operand
400     || matches!(self.peek_previous(), Some(t) if matches!(t.kind, TokenType::Identifier))
401     || matches!(self.peek_previous(), Some(t) if matches!(t.kind, TokenType::RightParenthesis
402 ))
403     || matches!(self.peek_previous(), Some(t) if matches!(t.kind, TokenType::Number));
404     if !allow {
405         self.errors
406             .push(syntax_error!(UnexpectedParenthesis, token));
407     }

```

```

407
408     if let Some(previous) = self.peek_previous()
409     && matches!(previous.kind, TokenType::Number)
410     {
411         // Function name cannot start with a number
412         self.errors
413         .push(syntax_error!(InvalidFunctionName, previous));
414     }
415
416     if let Some(previous) = self.peek_previous()
417     && matches!(previous.kind, TokenType::RightParenthesis)
418     {
419         // Needed operation. but anyway, pushing to the stack
420         self.errors
421         .push(syntax_error!(UnexpectedParenthesis, token));
422     }
423
424     self.parentheses_stack.push_back(token.clone());
425     self.status.expect_operand = true;
426     self.status.expect_operator = false;
427     self.current_index += 1;
428     continue;
429 },
430
431 TokenType::RightParenthesis => {
432     // Empty grouping check. Also, empty function is not an error.
433     if let Some(possible_left_parentheses) = self.peek_previous()
434     && matches!(
435         possible_left_parentheses.kind,
436         TokenType::LeftParenthesis
437     )
438     {
439         // But, non-function
440         if let Some(possible_function_name) = self.peek_previous_by(2)
441         && matches!(
442             possible_function_name.kind,
443             TokenType::Identifier
444         )
445         {
446             self.status.expect_operand = false;
447             self.status.expect_operator = true;
448         } else {
449             self.errors.push(syntax_error!(EmptyParentheses, token));
450         }
451     } else if self.status.expect_operand {
452         self.errors
453         .push(syntax_error!(UnexpectedParenthesis, token));
454     }
455
456     match self.parentheses_stack.pop_back().is_some() {

```

```

457     true => {
458         // Correct
459         self.status.expect_operand = false;
460         self.status.expect_operator = true;
461     },
462     false => {
463         self.errors.push(syntax_error!(UnmatchedParenthesis, token))
464     },
465 }
466
467 self.current_index += 1;
468 continue;
469 },
470
471 TokenType::Comma => {
472     // Allowed only inside parentheses (function)
473     if self.parentheses_stack.is_empty() {
474         // Surely an error
475         self.errors.push(syntax_error!(UnexpectedComma, token));
476         self.status.expect_operand = true;
477         self.status.expect_operator = false;
478         self.current_index += 1;
479         continue;
480     }
481
482     // Inside parentheses comma need to be after operand and before new operand
483     if self.status.expect_operand {
484         // Empty argument
485         self.errors.push(syntax_error!(UnexpectedComma, token));
486         self.current_index += 1;
487         continue;
488     }
489
490     // Argument is not present
491     if let Some(next) = self.peek_next()
492     && matches!(next.kind, TokenType::RightParenthesis)
493     {
494         // Empty argument
495         self.errors.push(syntax_error!(MissingArgument, token));
496         self.current_index += 1;
497         continue;
498     }
499
500     // Expecting new operand
501     self.status.expect_operand = true;
502     self.status.expect_operator = false;
503     self.current_index += 1;
504     continue;
505 },
506

```

```

507     TokenType::Unknown => {
508         // Unknown – always an error
509         self.errors.push(syntax_error!(UnknownToken, token));
510         self.current_index += 1;
511         continue;
512     },
513     TokenType::NewLine => {
514         // Unexpected newline is error, if we're not in string
515         if !self.status.in_string {
516             self.errors.push(syntax_error!(UnexpectedNewLine, token));
517         }
518         self.current_index += 1;
519         continue;
520     },
521     TokenType::Space | TokenType::Tab => {
522         // Shouldn't be here, but skipping just in case
523         self.current_index += 1;
524         continue;
525     },
526 }
527 }
528
529 // Error for every unmatched left parenthesis
530 for unmatched in self.parentheses_stack.into_iter() {
531     self.errors
532     .push(syntax_error!(UnmatchedParenthesis, unmatched));
533 }
534
535 // If operand is expected in the end, it's the error.
536 if let Some(last) = self.tokens.last()
537 && self.status.expect_operand
538 {
539     self.errors
540     .push(syntax_error!(UnexpectedEndOfExpression, last));
541 }
542
543 // Unclosed string
544 if let Some(token) = self.quotation_marks_stack.pop_back()
545 && self.status.in_string
546 {
547     self.errors
548     .push(syntax_error!(UnmatchedQuotationMark, token));
549 }
550
551 self.errors
552 .sort_by(|a, b| a.token.position.start.cmp(&b.token.position.start));
553
554 self.errors
555 }

```

```
557     fn peek_next(&self) -> Option<&Token> {
558         self.tokens.get(self.current_index + 1)
559     }
560
561     fn peek_next_by(&self, by: usize) -> Option<&Token> {
562         self.tokens.get(self.current_index + by)
563     }
564
565     fn peek_previous(&self) -> Option<&Token> {
566         self.tokens.get(self.current_index.checked_sub(1)?)
567     }
568
569     fn peek_previous_by(&self, by: usize) -> Option<&Token> {
570         self.tokens.get(self.current_index.checked_sub(by)?)
571     }
572 }
573
```