

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

Факультет інформатики та обчислювальної техніки
Кафедра обчислювальної техніки

ЗВІТ

з лабораторної роботи №2
з дисципліни ”Програмне забезпечення комп’ютерних систем”

Тема: Розпаралелювання арифметичного виразу

Варіант №5

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

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

Дата здачі: 04.11.2025

Мета роботи. Виконати автоматичне розпаралелювання арифметичного виразу.

Вхідні дані: Коректний арифметичний вираз після успішного виконання лексичного та синтаксичного аналізу (результат виконання лабораторної роботи №1).

Завдання: За аналітичним записом арифметичного виразу (АВ) побудувати його дерево паралельної форми максимальної ширини (максимальна кількість операцій в одному ярусі) та мінімальної довжини (мінімально можлива кількість ярусів). Для виконання цієї лабораторної роботи необхідно розробити алгоритм побудови дерева паралельної форми за записом АВ та реалізувати цей алгоритм на будь-якій мові програмування. Спосіб зображення отриманого дерева визначається студентом. Окремої уваги заслуговують фрагменти АВ типу A/S/D/F/G/H або A-S-D-F-G-H. У цих випадках для того, щоб розпаралелювання стало можливим можна використати один із двох способів:

1. Можна замінити фрагмент АВ A/S/D/F/G/H на $A/(S*D*F*G*H)$ або фрагмент A-S-D-F-G-H на $A-(S+D+F+G+H)$;
2. Можна частину операцій / замінити на *, а частину операцій – відповідно на +.

В обох випадках ми можемо досягти прискорення виконання заданого АВ за рахунок додаткового паралелізму, а також зменшення загальної трудомісткості обчислень, оскільки зменшується число операцій ділення і збільшується кількість операцій множення.

Хід роботи.

У ході виконання роботи було реалізовано повний конвеєр для автоматичного розпаралелювання арифметичних виразів. На відміну від першої лабораторної роботи, яка фокусувалася лише на виявленні лексичних та синтаксичних помилок, ця робота починається з коректного, синтаксично перевіреного потоку лексем.

Основою для всіх подальших трансформацій є абстрактне синтаксичне дерево (AST). Спочатку береться синтаксично коректний потік лексем (сформований `lexer.rs`) і перетворює його на деревовидну структуру. Для цього використовується `AstParser` – парсер, реалізований за методом рекурсивного спуску. Він послідовно обробляє лексеми, дотримуючись пріоритету операцій: `parse_logical_or` викликає `parse_logical_and`, той викликає `parse_expression` (для + та -), який, у свою чергу, викликає `parse_term` (для * та /), і так далі

аж до `parse_primary`. Цей парсер також коректно обробляє унарні операції, групування в дужках, виклики функцій та доступ до елементів масиву, будуючи ієрархічну структуру `AstNode`. Отримане початкове дерево, яке відображає пріоритет операцій вихідного виразу, є відправною точкою для подальших етапів оптимізації, трансформації та балансування.

Процес компіляції було організовано як багатоетапний конвеєр, визначений у файлі `compiler.rs`. Кожен етап виконує специфічну задачу трансформації або оптимізації дерева, передаючи результат наступному етапу.

Першим кроком після парсингу є початкова оптимізація, реалізована у модулі `math.rs`. Цей етап виконує рекурсивний обхід дерева знизу вгору. Його головна задача – обчислення констант (`constant folding`) та виконання базових алгебраїчних спрощень. Наприклад, вирази $2 + 2$ обчислюються в 4, а $A * 0$ – в 0.0. Було реалізовано правила для спрощення операцій з нулем та одиницею, такі як $A + 0$ (стає A), $A * 1$ (стає A), $0 / A$ (стає 0.0) тощо. Також було додано потужніше правило для обробки ідентичних піддерев: вирази $A - A$ спрощуються до 0.0, а A / A – до 1.0. Цей етап також відповідає за виявлення семантичних помилок, таких як ділення на нуль, включаючи складні випадки, як $(A*0) / (B-B)$. Якщо вираз повністю обчислюється до єдиного числа, подальша обробка припиняється.

Демонстрація згортання констант:

The screenshot shows an IDE interface with a project named 'KPI-SCS'. The main editor displays the expression `10+(a-b)/(a-b)+13`. Below the editor, the 'Terminal' tab shows the output of the 'Abstract-Syntax Tree generation' process, which is successful. The terminal output displays the following tree structure:

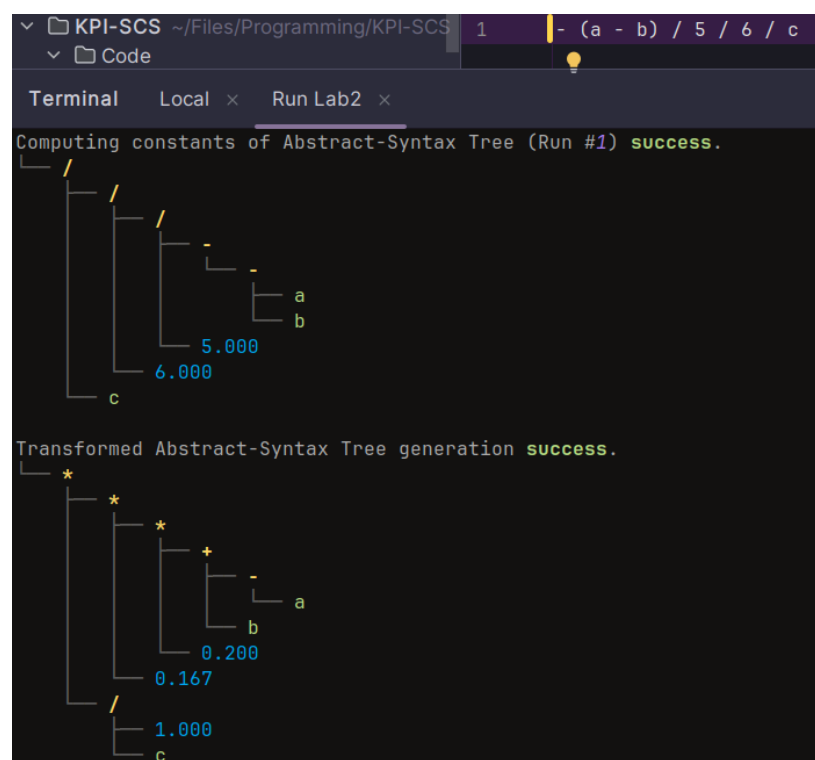
```
Abstract-Syntax Tree generation success.
├── +
│   ├── 10.000
│   └── /
│       ├── -
│       │   ├── a
│       │   └── b
│       └── -
│           ├── a
│           └── b
└── 13.000
```

Below the tree, the terminal shows the result of computing constants: 'Computing constants of Abstract-Syntax Tree (Run #1) success.' followed by the final value '24.000'.

Наступним і ключовим етапом є трансформація дерева, реалізована в `transform.rs`. Мета цього модуля – позбутися операцій, які заважають розпаралелюванню (бінарні віднімання та ділення) і підготувати дерево до балансування. Це досягається шляхом заміни цих операцій на їхні асоціативні еквіваленти. Було реалізовано два основних правила. По-перше, операція віднімання $A - B$ перетворюється на операцію додавання з унарним мінусом: $A + (-B)$. По-друге, операція ділення A / B перетворюється на множення: $A * (1.0 / B)$.

Найскладнішою частиною етапу трансформації є коректна обробка дужок, особливо тих, перед якими стоїть знак мінус. Це реалізовано через рекурсивну логіку для унарної операції ”мінус”. Коли `transform_recursive` зустрічає унарний мінус, він аналізує вираз під ним. Якщо це інший унарний мінус, спрацьовує правило $-(-A) \rightarrow A$, і подвійний мінус скорочується. Якщо це бінарна операція, правила розкриття дужок застосовуються рекурсивно: $-(A + B)$ перетворюється на $(-A) + (-B)$, а $-(A - B)$ – на $(-A) + B$. Для запуску цього механізму, сама трансформація $A - B$ рекурсивно викликає `transform_recursive` на своєму результаті $A + (-B)$. Також було додано оптимізацію для ”проштовхування” мінуса до констант: $-(Num * B)$ перетворюється на $(-Num) * B$. Щоб уникнути нескінченної рекурсії або подвійної трансформації (що було виявлено під час тестування), до правила ділення було додано ”ідемпотентну” перевірку: якщо вираз вже має вигляд $1.0 / B$, він не трансформується повторно.

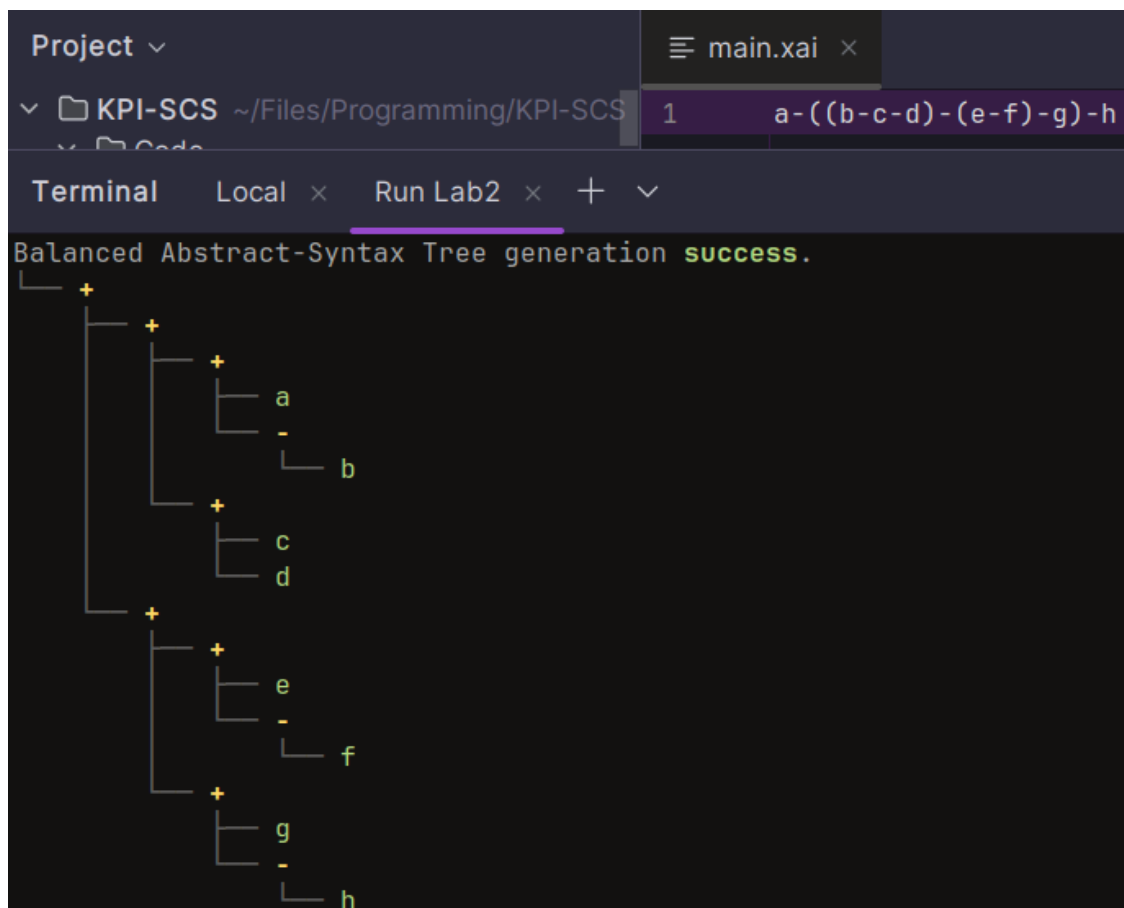
Демонстрація операції трансформації:



Після трансформації дерево може містити нові константні вирази (наприклад, $A / 2.0$ стало $A * 0.5$, або $B / 1.0$ стало $B * 1.0$). Тому конвеєр запускає другий етап оптимізації згортання констант. Цей прохід, використовуючи ту саму логіку з `math.rs`, "підчищає" дерево, спрощуючи $B * 1.0$ до B і об'єднуючи нові константи, які могли опинитися поруч.

Останнім основним етапом є балансування дерева, реалізоване в `balancer.rs`. Цей модуль приймає трансформоване і спрощене дерево (яке тепер складається переважно з ланцюжків асоціативних операцій $+$ та $*$) і будує з нього дерево мінімальної висоти. Алгоритм працює у два кроки. Спочатку функція `collect_operands` рекурсивно "сплощує" ланцюжки однакових асоціативних операцій (наприклад, дерево $((a + (-b)) + c)$ стає списком $[a, (-b), c]$). Потім функція `build_balanced_tree` будує збалансоване дерево з цього списку. Вона використовує `VecDeque` як чергу: всі операнди додаються до черги. Далі в циклі, доки в черзі більше одного елемента, вузли попарно дістаються з початку черги, об'єднуються в новий `BinaryOperation` (наприклад, $left + right$), і цей новий батьківський вузол кладеться в кінець черги. Цей процес повторюється, створюючи яруси дерева знизу вгору, і гарантує, що кінцеве дерево матиме мінімально можливу висоту.

Демонстрація балансування:



Нарешті, конвеєр виконує третій, фінальний запуск згортання констант. Він потрібен для рідкісних випадків, коли процес балансування могло випадково поставити два обчислюваних вузли поруч, дозволяючи останнє фінальне спрощення. У результаті роботи цього конвеєра вхідний арифметичний вираз перетворюється на збалансоване, алгебраїчно спрощене дерево, повністю готове до паралельного виконання.

Висновок. Під час виконання лабораторної роботи було успішно досягнуто її мети – автоматичного розпаралелювання арифметичних виразів . Для цього було реалізовано багатоетапний конвеєр компіляції, що включає три ключові модулі.

Лістинг.

compiler.rs

```
1 use crate::compiler::ast::tree::{AbstractSyntaxTree, AstParser};
2 use crate::compiler::lexer::Lexer;
3 use crate::compiler::syntax::SyntaxAnalyzer;
4
5 pub fn compile(source: &str, is_pretty: bool) {
6     // Lexical Analysis
7     let tokens = tokenizer::tokenize(source);
8     // Syntax Analysis
9     let syntax_errors = SyntaxAnalyzer::new(&tokens).analyze();
10    let is_syntax_analysis_successful = syntax_errors.is_empty();
11    syntax::report(source, syntax_errors, is_pretty);
12    if !is_syntax_analysis_successful {
13        return;
14    }
15
16    // Making lexemes
17    let lexemes_result = Lexer::new(tokens).run();
18    let lexemes = match lexemes_result {
19        Ok(lexemes) => {
20            lexer::report_success(&lexemes);
21            lexemes
22        },
23        Err(error) => {
24            lexer::report_error(error);
25            return;
26        },
27    };
28
29    // AST Generation
30    let ast_result = AstParser::new(lexemes).parse();
31    let ast = match ast_result {
32        Ok(ast) => {
33            ast::tree::report_success(&ast);
34            ast
35        },
36        Err(error) => {
37            ast::tree::report_error(error);
38            return;
39        },
40    };
41    // AST Math Optimization, #1
42    let ast = match compute_run(ast, 1) {
43        Some(ast) => ast,
44        None => return,
45    };
46    // AST Parallelization
47    let ast_result = ast.transform();
```

```

48     let ast = match ast_result {
49         Ok(ast) => {
50             ast::transform::report_success(&ast);
51             ast
52         },
53         Err(error) => {
54             ast::transform::report_error(error);
55             return;
56         },
57     };
58     // AST Math Optimization, #2
59     let ast = match compute_run(ast, 2) {
60         Some(ast) => ast,
61         None => return,
62     };
63     // AST Balancing
64     let ast_result = ast.balance();
65     let ast = match ast_result {
66         Ok(ast) => {
67             ast::balancer::report_success(&ast);
68             ast
69         },
70         Err(error) => {
71             ast::balancer::report_error(error);
72             return;
73         },
74     };
75     // AST Math Optimization, #3
76     let _ast = match compute_run(ast, 3) {
77         Some(ast) => ast,
78         None => return,
79     };
80 }
81
82 fn compute_run(tree: AbstractSyntaxTree, number: u8) -> Option<AbstractSyntaxTree> {
83     // AST Math Optimization
84     let ast_result = tree.compute();
85     let ast = match ast_result {
86         Ok(ast) => {
87             ast::math::report_success(&ast, number);
88             ast
89         },
90         Err(error) => {
91             ast::math::report_error(error, number);
92             return None;
93         },
94     };
95     if ast::math::check_finalization(&ast) {
96         return None;
97     }

```



```

98     Some(ast)
99 }
100
101 pub mod ast {
102     pub mod balancer;
103     pub mod math;
104     pub mod transform;
105     pub mod tree;
106 }
107 pub mod lexer;
108 pub mod syntax;
109 pub mod tokenizer;
110

```

tree.rs

```

1  use crate::compiler::lexer::Lexeme;
2  use colored::Colorize;
3
4  #[derive(Debug, Clone, PartialEq)]
5  pub struct AbstractSyntaxTree {
6      pub peek: AstNode,
7  }
8
9  impl AbstractSyntaxTree {
10     pub fn from_node(node: AstNode) -> Self {
11         Self { peek: node }
12     }
13
14     pub fn pretty_print(&self) -> String {
15         let mut tree = String::new();
16         Self::print_recursive(&self.peek, &mut tree, "".to_string(), true);
17         tree
18     }
19
20     fn print_recursive(node: &AstNode, tree: &mut String, prefix: String, is_last: bool) {
21         let connector = if is_last { "└─ " } else { "├─ " };
22
23         tree.push_str(&format!("{:?}", prefix.dimmed(), connector.dimmed()));
24
25         let node_text = match node {
26             AstNode::Number(n) => format!("{n:.3}").bright_blue(),
27             AstNode::Identifier(s) => s.to_string().green(),
28             AstNode::StringLiteral(s) => format!("{:?}", s).bright_magenta(),
29             AstNode::UnaryOperation { operation, .. } => {
30                 operation.to_string().yellow().bold()
31             },
32             AstNode::BinaryOperation { operation, .. } => {
33                 operation.to_string().yellow().bold()
34             },
35             AstNode::FunctionCall { name, .. } => format!("{:?}", name).cyan().bold(),

```

```

36     AstNode::ArrayAccess { identifier, .. } => {
37         format!("{}[...]", identifier).blue().bold()
38     },
39 };
40 tree.push_str(&format!("{ }\n", node_text));
41
42 let new_prefix = prefix + if is_last { "    " } else { "|    " };
43
44 match node {
45     AstNode::Number(_) | AstNode::Identifier(_) | AstNode::StringLiteral(_) => {},
46
47     AstNode::UnaryOperation { expression, .. } => {
48         Self::print_recursive(expression, tree, new_prefix, true);
49     },
50
51     AstNode::BinaryOperation { left, right, .. } => {
52         Self::print_recursive(left, tree, new_prefix.clone(), false);
53         Self::print_recursive(right, tree, new_prefix, true);
54     },
55
56     AstNode::FunctionCall { arguments, .. } => {
57         let arg_count = arguments.len();
58         for (i, arg) in arguments.iter().enumerate() {
59             let is_last_arg = i == arg_count - 1;
60             Self::print_recursive(arg, tree, new_prefix.clone(), is_last_arg);
61         }
62     },
63
64     AstNode::ArrayAccess {
65         identifier: _,
66         indices,
67     } => {
68         let dimensions = indices.len();
69         for (i, index) in indices.iter().enumerate() {
70             let is_last_arg = i == dimensions - 1;
71             Self::print_recursive(index, tree, new_prefix.clone(), is_last_arg);
72         }
73     },
74 }
75 }
76 }
77
78 #[derive(Debug, Clone, PartialEq)]
79 pub enum AstNode {
80     Number(f64),
81     Identifier(String),
82     StringLiteral(String),
83     UnaryOperation {
84         operation: UnaryOperationKind,
85         expression: Box<AstNode>,

```

```

86     },
87     BinaryOperation {
88         operation: BinaryOperationKind,
89         left: Box<AstNode>,
90         right: Box<AstNode>,
91     },
92     FunctionCall {
93         name: String,
94         arguments: Vec<AstNode>,
95     },
96     ArrayAccess {
97         identifier: String,
98         indices: Vec<AstNode>,
99     },
100 }
101
102 #[derive(Debug, Clone, PartialEq)]
103 pub enum UnaryOperationKind {
104     Minus,
105     Not,
106 }
107
108 #[derive(Debug, Clone, PartialEq)]
109 pub enum BinaryOperationKind {
110     Plus,
111     Minus,
112     Multiply,
113     Divide,
114     Or,
115     And,
116 }
117
118 pub struct AstParser {
119     lexemes: Vec<Lexeme>,
120     current_index: usize,
121 }
122
123 impl AstParser {
124     pub fn new(lexemes: Vec<Lexeme>) -> Self {
125         Self {
126             lexemes,
127             current_index: 0,
128         }
129     }
130
131     pub fn parse(&mut self) -> Result<AbstractSyntaxTree, AstError> {
132         let node = self.parse_logical_or()?;
133
134         if self.peek().is_some()
135         && let Some(peek) = self.consume()

```

```

136     {
137         Err(AstError::NotExpectedLexeme(peek.clone()))
138     } else {
139         Ok(AbstractSyntaxTree { peek: node })
140     }
141 }
142
143 fn parse_logical_or(&mut self) -> Result<AstNode, AstError> {
144     let mut left_node = self.parse_logical_and()?;
145
146     while let Some(Lexeme::Or) = self.peek()
147     && let Some(_) = self.consume()
148     {
149         let right_node = self.parse_logical_and()?;
150         left_node = AstNode::BinaryOperation {
151             operation: BinaryOperationKind::Or,
152             left: Box::new(left_node),
153             right: Box::new(right_node),
154         };
155     }
156     Ok(left_node)
157 }
158
159 fn parse_logical_and(&mut self) -> Result<AstNode, AstError> {
160     let mut left_node = self.parse_expression()?;
161
162     while let Some(Lexeme::And) = self.peek()
163     && let Some(_) = self.consume()
164     {
165         let right_node = self.parse_expression()?;
166         left_node = AstNode::BinaryOperation {
167             operation: BinaryOperationKind::And,
168             left: Box::new(left_node),
169             right: Box::new(right_node),
170         };
171     }
172     Ok(left_node)
173 }
174
175 fn parse_expression(&mut self) -> Result<AstNode, AstError> {
176     let mut left_node = self.parse_term()?;
177
178     while let Some(Lexeme::Plus) | Some(Lexeme::Minus) = self.peek()
179     && let Some(lexeme) = self.consume()
180     {
181         let operation = match lexeme {
182             Lexeme::Plus => BinaryOperationKind::Plus,
183             Lexeme::Minus => BinaryOperationKind::Minus,
184             _ => return Err(AstError::UnreachableLexeme(lexeme.clone())),
185         };

```

```

186
187     let right_node = self.parse_term()?;
188
189     left_node = AstNode::BinaryOperation {
190         operation,
191         left: Box::new(left_node),
192         right: Box::new(right_node),
193     };
194 }
195
196 Ok(left_node)
197 }
198
199 fn parse_term(&mut self) -> Result<AstNode, AstError> {
200     let mut left_node = self.parse_unary()?;
201
202     while let Some(Lexeme::Multiply) | Some(Lexeme::Divide) = self.peek()
203     && let Some(lexeme) = self.consume()
204     {
205         let operation = match lexeme {
206             Lexeme::Multiply => BinaryOperationKind::Multiply,
207             Lexeme::Divide => BinaryOperationKind::Divide,
208             _ => return Err(AstError::UnreachableLexeme(lexeme.clone())),
209         };
210
211         let right_node = self.parse_unary()?;
212
213         left_node = AstNode::BinaryOperation {
214             operation,
215             left: Box::new(left_node),
216             right: Box::new(right_node),
217         };
218     }
219
220     Ok(left_node)
221 }
222
223 fn parse_unary(&mut self) -> Result<AstNode, AstError> {
224     if let Some(Lexeme::Not) | Some(Lexeme::Minus) = self.peek()
225     && let Some(lexeme) = self.consume()
226     {
227         let operation_kind = match lexeme {
228             Lexeme::Not => UnaryOperationKind::Not,
229             Lexeme::Minus => UnaryOperationKind::Minus,
230             _ => return Err(AstError::UnreachableLexeme(lexeme.clone())),
231         };
232
233         let child_node = self.parse_unary()?;
234
235         Ok(AstNode::UnaryOperation {

```

```

236         operation: operation_kind,
237         expression: Box::new(child_node),
238     })
239 } else {
240     self.parse_primary()
241 }
242 }
243
244 fn parse_primary(&mut self) -> Result<AstNode, AstError> {
245     if let Some(lexeme) = self.consume() {
246         match lexeme {
247             Lexeme::Number(value) => Ok(AstNode::Number(value)),
248             Lexeme::String(value) => {
249                 match (matches!(self.peak(), Some(Lexeme::Comma)))
250                     || (matches!(self.peak_previous_by(2), Some(Lexeme::Comma)))
251                 {
252                     true => Ok(AstNode::StringLiteral(value.clone())),
253                     false => Err(AstError::StringOutsideFunction(value.clone())),
254                 }
255             },
256
257             Lexeme::LeftParenthesis => {
258                 let inner_node = self.parse_logical_or()?;
259
260                 if self.peak() == Some(&Lexeme::RightParenthesis) {
261                     self.consume();
262                     Ok(inner_node)
263                 } else {
264                     Err(AstError::ExpectedRightParenthesis)
265                 }
266             },
267
268             Lexeme::Identifier(name) => {
269                 if self.peak() == Some(&Lexeme::LeftParenthesis)
270                 && let Some(_) = self.consume()
271                 {
272                     let function_name = name.clone();
273                     let mut args = Vec::new();
274
275                     if self.peak() != Some(&Lexeme::RightParenthesis) {
276                         loop {
277                             args.push(self.parse_logical_or()?);
278
279                             let peek = self.peak();
280
281                             if peek == Some(&Lexeme::Comma) {
282                                 let _ = self.consume();
283                             } else if peek == Some(&Lexeme::RightParenthesis) {
284                                 break;
285                             } else {

```

```

286         return Err(match peek {
287             None => AstError::NotExpectedEndOfExpression,
288             Some(lexeme) => {
289                 AstError::ExpectedCommaOrRightParenthesis(
290                     lexeme.clone(),
291                 )
292             },
293         });
294     }
295 }
296 }
297
298 let _ = self.consume();
299
300 Ok(AstNode::FunctionCall {
301     name: function_name,
302     arguments: args,
303 })
304 } else if self.peek() == Some(&Lexeme::LeftBracket) {
305     let identifier = name.clone();
306     let mut indices: Vec<AstNode> = Vec::new();
307
308     loop {
309         let _ = self.consume();
310         let index = self.parse_logical_or()?;
311         if self.peek() == Some(&Lexeme::RightBracket) {
312             let _ = self.consume();
313             indices.push(index);
314             if self.peek() == Some(&Lexeme::LeftBracket) {
315                 continue;
316             } else {
317                 break;
318             }
319         } else {
320             return Err(AstError::ExpectedRightBracket);
321         }
322     }
323     Ok(AstNode::ArrayAccess {
324         identifier,
325         indices,
326     })
327 } else {
328     Ok(AstNode::Identifier(name.clone()))
329 }
330 },
331
332 _ => Err(AstError::NotExpectedLexeme(lexeme.clone())),
333 }
334 } else {
335     Err(AstError::NotExpectedEndOfExpression)

```

```

336     }
337 }
338
339 fn consume(&mut self) -> Option<Lexeme> {
340     if let Some(lexeme) = self.peek() {
341         let lexeme = lexeme.clone();
342         self.current_index += 1;
343         return Some(lexeme);
344     }
345     None
346 }
347
348 fn peek(&self) -> Option<&Lexeme> {
349     self.lexemes.get(self.current_index)
350 }
351
352 fn peek_previous_by(&self, by: usize) -> Option<&Lexeme> {
353     self.lexemes.get(self.current_index - by)
354 }
355 }
356
357 pub fn report_success(tree: &AbstractSyntaxTree) {
358     log::warn!(
359         "{} {}. ",
360         "Abstract-Syntax Tree generation",
361         "success".bold().green()
362     );
363     log::info!("{}", tree.pretty_print());
364 }
365
366 pub fn report_error(error: AstError) {
367     log::error!("{}", "AST error:".bold().red(), error);
368 }
369
370 #[derive(Debug, PartialEq)]
371 pub enum AstError {
372     ExpectedRightBracket,
373     ExpectedRightParenthesis,
374     ExpectedCommaOrRightParenthesis(Lexeme),
375     NotExpectedEndOfExpression,
376     NotExpectedLexeme(Lexeme),
377     StringOutsideFunction(String),
378     UnreachableLexeme(Lexeme),
379
380     CannotBuildEmptyTree,
381     FailedPopFromQueue,
382     DivisionByZero(AstNode),
383 }
384
385 impl std::fmt::Display for AstError {

```



```

386 fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
387     let text = match self {
388         Self::ExpectedCommaOrRightParenthesis(lexeme) => &format!(
389             "Expected ',', ' or ')", but found "{}\\"",
390             lexeme.display_type()
391         ),
392         Self::ExpectedRightBracket => "Expected right bracket.",
393         Self::ExpectedRightParenthesis => "Expected right parenthesis.",
394         Self::NotExpectedEndOfExpression => "Not expected end of expression.",
395         Self::NotExpectedLexeme(lexeme) => {
396             &format!("Not expected lexeme '{}\\"", lexeme.display_type())
397         },
398         Self::StringOutsideFunction(string) => {
399             &format!("String literal '{}\" outside function call.", string)
400         },
401         Self::UnreachableLexeme(lexeme) => {
402             &format!("Unreachable lexeme '{}\\"", lexeme.display_type())
403         },
404
405         Self::CannotBuildEmptyTree => {
406             "Cannot build a balanced tree from zero operands"
407         },
408         Self::FailedPopFromQueue => {
409             "Failed to pop node from the queue during tree construction"
410         },
411         Self::DivisionByZero(node) => &format!("Division by zero. Node: {:#?}", node),
412     };
413
414     write!(f, "{}", text)
415 }
416 }
417
418 impl std::fmt::Display for UnaryOperationKind {
419     fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
420         match self {
421             Self::Minus => write!(f, "-"),
422             Self::Not => write!(f, "!"),
423         }
424     }
425 }
426
427 impl std::fmt::Display for BinaryOperationKind {
428     fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
429         match self {
430             Self::Plus => write!(f, "+"),
431             Self::Minus => write!(f, "-"),
432             Self::Multiply => write!(f, "*"),
433             Self::Divide => write!(f, "/"),
434             Self::Or => write!(f, "|"),
435             Self::And => write!(f, "&"),

```

```

436     }
437 }
438 }
439

```

math.rs

```

1  use crate::compiler::ast::tree::{
2      AbstractSyntaxTree, AstError, AstNode, BinaryOperationKind, UnaryOperationKind,
3  };
4  use colored::Colorize;
5
6  impl AbstractSyntaxTree {
7      pub fn compute(self) -> Result<AbstractSyntaxTree, AstError> {
8          let computed = Self::compute_recursive(self.peek)?;
9
10         Ok(Self::from_node(computed))
11     }
12
13     fn compute_recursive(node: AstNode) -> Result<AstNode, AstError> {
14         match &node {
15             AstNode::Number(_) | AstNode::Identifier(_) | AstNode::StringLiteral(_) => {
16                 Ok(node)
17             },
18             AstNode::UnaryOperation {
19                 operation: op,
20                 expression,
21             } => match &op {
22                 UnaryOperationKind::Minus => {
23                     let child = Self::compute_recursive(*expression.clone())?;
24                     if let AstNode::Number(number) = child {
25                         Ok(AstNode::Number(-number))
26                     } else {
27                         Ok(node)
28                     }
29                 },
30                 UnaryOperationKind::Not => Ok(node),
31             },
32             AstNode::BinaryOperation {
33                 operation,
34                 left,
35                 right,
36             } => match operation {
37                 BinaryOperationKind::Plus
38                 | BinaryOperationKind::Minus
39                 | BinaryOperationKind::Multiply
40                 | BinaryOperationKind::Divide => {
41                     let computed_left = Self::compute_recursive(*left.clone())?;
42                     let computed_right = Self::compute_recursive(*right.clone())?;
43
44                     // Case: (a + b) - (a + b) = 0

```

```

45 // Or: (a + b) / (a + b) = 1
46 if computed_left.eq(&computed_right) {
47     match operation {
48         BinaryOperationKind::Minus => {
49             return Ok(AstNode::Number(0.0));
50         },
51         BinaryOperationKind::Divide => {
52             if let AstNode::Number(number) = &computed_left
53             && *number == 0.0
54             {
55                 // Case: (5 - 5) / (5 - 5)
56                 return Err(AstError::DivisionByZero(node));
57             }
58             return Ok(AstNode::Number(1.0));
59         },
60         _ => {},
61     }
62 }
63
64 if let (AstNode::Number(left_number), AstNode::Number(right_number)) =
65 (&computed_left, &computed_right)
66 {
67     let result = match operation {
68         BinaryOperationKind::Plus => left_number + right_number,
69         BinaryOperationKind::Minus => left_number - right_number,
70         BinaryOperationKind::Multiply => left_number * right_number,
71         BinaryOperationKind::Divide => {
72             if *right_number == 0.0 {
73                 return Err(AstError::DivisionByZero(node));
74             } else {
75                 left_number / right_number
76             }
77         },
78         _ => unreachable!(),
79     };
80     Ok(AstNode::Number(result))
81 } else if let AstNode::Number(number) = &computed_left {
82     if number == &0.0 {
83         if [
84             BinaryOperationKind::Multiply,
85             BinaryOperationKind::Divide,
86         ]
87         .contains(operation)
88         {
89             return Ok(AstNode::Number(0.0));
90         }
91         if BinaryOperationKind::Plus == *operation {
92             return Ok(computed_right);
93         }
94         if BinaryOperationKind::Minus == *operation {

```

```

95         return Ok(AstNode::UnaryOperation {
96             operation: UnaryOperationKind::Minus,
97             expression: Box::new(computed_right),
98         });
99     }
100 }
101 if number == &1.0 && BinaryOperationKind::Multiply == *operation {
102     return Ok(computed_right);
103 }
104
105 Ok(AstNode::BinaryOperation {
106     operation: operation.clone(),
107     left: Box::new(computed_left),
108     right: Box::new(computed_right),
109 })
110 } else if let AstNode::Number(number) = &computed_right {
111     if number == &0.0 {
112         if BinaryOperationKind::Divide == *operation {
113             return Err(AstError::DivisionByZero(node));
114         }
115         if BinaryOperationKind::Multiply == *operation {
116             return Ok(AstNode::Number(0.0));
117         }
118         if [BinaryOperationKind::Plus, BinaryOperationKind::Minus]
119             .contains(operation)
120         {
121             return Ok(computed_left);
122         }
123     }
124     if number == &1.0 && BinaryOperationKind::Multiply == *operation {
125         return Ok(computed_left);
126     }
127
128     Ok(AstNode::BinaryOperation {
129         operation: operation.clone(),
130         left: Box::new(computed_left),
131         right: Box::new(computed_right),
132     })
133 } else {
134     Ok(AstNode::BinaryOperation {
135         operation: operation.clone(),
136         left: Box::new(computed_left),
137         right: Box::new(computed_right),
138     })
139 }
140 },
141 _ => Ok(node),
142 },
143 AstNode::FunctionCall { name, arguments } => {
144     let mut computed_arguments = Vec::new();

```

```

145     for arg in arguments {
146         let arg = Self::compute_recursive(arg.clone())?;
147         computed_arguments.push(arg);
148     }
149
150     Ok(AstNode::FunctionCall {
151         name: name.clone(),
152         arguments: computed_arguments,
153     })
154 },
155 AstNode::ArrayAccess {
156     identifier,
157     indices,
158 } => {
159     let mut computed_indices = Vec::new();
160     for index in indices {
161         let index = Self::compute_recursive(index.clone())?;
162         computed_indices.push(index);
163     }
164     Ok(AstNode::ArrayAccess {
165         identifier: identifier.clone(),
166         indices: computed_indices,
167     })
168 },
169 }
170 }
171 }
172
173 pub fn report_success(tree: &AbstractSyntaxTree, run: u8) {
174     log::warn!(
175         "Computing constants of Abstract-Syntax Tree (Run #{}) {}.",
176         run.to_string().bright_magenta().italic(),
177         "success".bold().green()
178     );
179     log::info!("{}", tree.pretty_print());
180 }
181
182 pub fn report_error(error: AstError, run: u8) {
183     log::error!(
184         "{} (Run #{}) {}",
185         "Computing constants of Abstract-Syntax Tree:".bold().red(),
186         run.to_string().bright_magenta().italic(),
187         error
188     );
189 }
190
191 pub fn check_finalization(tree: &AbstractSyntaxTree) -> bool {
192     if let AstNode::Number(number) = &tree.peak {
193         log::warn!(
194             "{} = {}. {}.",

```

```

195     "Computing solved code, result".bold().green(),
196     number,
197     "Further optimization is not needed".bold().red(),
198 );
199 return true;
200 }
201 false
202 }
203

```

transform.rs

```

1 use crate::compiler::ast::tree::{
2     AbstractSyntaxTree, AstError, AstNode, BinaryOperationKind, UnaryOperationKind,
3 };
4 use colored::Colorize;
5
6 impl AbstractSyntaxTree {
7     pub fn transform(self) -> Result<AbstractSyntaxTree, AstError> {
8         let peek = Self::transform_recursive(self.peek)?;
9
10        Ok(Self::from_node(peek))
11    }
12
13    pub fn transform_recursive(node: AstNode) -> Result<AstNode, AstError> {
14        match &node {
15            AstNode::Number(_) | AstNode::Identifier(_) | AstNode::StringLiteral(_) => {
16                Ok(node)
17            },
18
19            AstNode::UnaryOperation {
20                operation,
21                expression,
22            } => {
23                let transformed_expression =
24                    Self::transform_recursive(*expression.clone())?;
25                match operation {
26                    UnaryOperationKind::Not => Ok(AstNode::UnaryOperation {
27                        operation: operation.clone(),
28                        expression: Box::new(transformed_expression),
29                    }),
30
31                    UnaryOperationKind::Minus => {
32                        match &transformed_expression {
33                            // Rule: -(-A) => A
34                            AstNode::UnaryOperation {
35                                operation: UnaryOperationKind::Minus,
36                                expression: inner_expr,
37                            } => Ok(*inner_expr.clone()),
38
39                            // Rule: -(A + B) => (-A) + (-B)

```

```

40     AstNode::BinaryOperation {
41         operation: BinaryOperationKind::Plus,
42         left,
43         right,
44     } => {
45         let new_left = AstNode::UnaryOperation {
46             operation: UnaryOperationKind::Minus,
47             expression: left.clone(),
48         };
49         let new_right = AstNode::UnaryOperation {
50             operation: UnaryOperationKind::Minus,
51             expression: right.clone(),
52         };
53         Self::transform_recursive(AstNode::BinaryOperation {
54             operation: BinaryOperationKind::Plus,
55             left: Box::new(new_left),
56             right: Box::new(new_right),
57         })
58     },
59
60     // Rule: -(A - B) => (-A) + B
61     AstNode::BinaryOperation {
62         operation: BinaryOperationKind::Minus,
63         left,
64         right,
65     } => {
66         let new_left = AstNode::UnaryOperation {
67             operation: UnaryOperationKind::Minus,
68             expression: left.clone(),
69         };
70         Self::transform_recursive(AstNode::BinaryOperation {
71             operation: BinaryOperationKind::Plus,
72             left: Box::new(new_left),
73             right: right.clone(),
74         })
75     },
76
77     // Rule: -(Num * B) => -Num * B
78     AstNode::BinaryOperation {
79         operation: BinaryOperationKind::Multiply,
80         left,
81         right,
82     } => {
83         if let AstNode::Number(number) = *left.clone() {
84             return Ok(AstNode::BinaryOperation {
85                 operation: BinaryOperationKind::Multiply,
86                 left: Box::new(AstNode::Number(-number)),
87                 right: Box::new(*right.clone()),
88             });
89         }

```

```

90
91         Ok(AstNode::UnaryOperation {
92             operation: UnaryOperationKind::Minus,
93             expression: Box::new(transformed_expression),
94         })
95     },
96
97     // Other cases (for example, -(A*B) or just -A)
98     // just leaving them (for example -(A*B) or just -A)
99     _ => Ok(AstNode::UnaryOperation {
100         operation: UnaryOperationKind::Minus,
101         expression: Box::new(transformed_expression),
102     }),
103 }
104 },
105 }
106 },
107
108 AstNode::FunctionCall { name, arguments } => {
109     let mut transformed_arguments = vec![];
110     for argument in arguments {
111         transformed_arguments
112             .push(Self::transform_recursive(argument.clone())?);
113     }
114
115     Ok(AstNode::FunctionCall {
116         name: name.clone(),
117         arguments: transformed_arguments,
118     })
119 },
120
121 AstNode::BinaryOperation {
122     operation,
123     left,
124     right,
125 } => {
126     let transformed_left = Self::transform_recursive(*left.clone())?;
127     let transformed_right = Self::transform_recursive(*right.clone())?;
128
129     match operation {
130         // Rule 1: A - B => A + (-B)
131         BinaryOperationKind::Minus => {
132             match &transformed_right {
133                 AstNode::Number(number) if number.is_sign_negative() => {
134                     // Rule 1: A - (-B) => A + B
135                     Ok(AstNode::BinaryOperation {
136                         operation: BinaryOperationKind::Plus,
137                         left: Box::new(transformed_left),
138                         right: Box::new(AstNode::Number(f64::abs(*number))),
139                     })

```



```

140     },
141     AstNode::Number(number) => {
142         // Rule 1: A - B => A + (-B)
143         Ok(AstNode::BinaryOperation {
144             operation: BinaryOperationKind::Plus,
145             left: Box::new(transformed_left),
146             right: Box::new(AstNode::Number(-number)),
147         })
148     },
149     _ => {
150         let new_right = AstNode::UnaryOperation {
151             operation: UnaryOperationKind::Minus,
152             expression: Box::new(transformed_right),
153         };
154
155         let result_node = AstNode::BinaryOperation {
156             operation: BinaryOperationKind::Plus,
157             left: Box::new(transformed_left),
158             right: Box::new(new_right),
159         };
160         Self::transform_recursive(result_node)
161     },
162 }
163 },
164
165 // Rule 2: A / B => A * (1 / B)
166 BinaryOperationKind::Divide => {
167     if let AstNode::Number(number) = transformed_left
168     && number == 1.0
169     {
170         return Ok(AstNode::BinaryOperation {
171             operation: BinaryOperationKind::Divide,
172             left: Box::new(transformed_left),
173             right: Box::new(transformed_right),
174         });
175     }
176
177     if let AstNode::Number(number) = transformed_right {
178         return if number == 0.0 {
179             Err(AstError::DivisionByZero(node))
180         } else {
181             Ok(AstNode::BinaryOperation {
182                 operation: BinaryOperationKind::Multiply,
183                 left: Box::new(transformed_left),
184                 right: Box::new(AstNode::Number(1.0 / number)),
185             })
186         };
187     }
188     Ok(AstNode::BinaryOperation {
189         operation: BinaryOperationKind::Multiply,

```

```

190         left: Box::new(transformed_left),
191         right: Box::new(AstNode::BinaryOperation {
192             operation: BinaryOperationKind::Divide,
193             left: Box::new(AstNode::Number(1.0)), // "1"
194             right: Box::new(transformed_right), // "B"
195         }),
196     })
197 },
198
199     // Other operations (Plus, Multiply, And, Or)
200     // left without editing, but with transformed kids.
201     _ => Ok(AstNode::BinaryOperation {
202         operation: operation.clone(),
203         left: Box::new(transformed_left),
204         right: Box::new(transformed_right),
205     }),
206 }
207 },
208
209     AstNode::ArrayAccess {
210         identifier,
211         indices,
212     } => {
213         let mut transformed_indices = vec![];
214         for index in indices {
215             transformed_indices.push(Self::transform_recursive(index.clone())?);
216         }
217         Ok(AstNode::ArrayAccess {
218             identifier: identifier.clone(),
219             indices: transformed_indices,
220         })
221     },
222 }
223 }
224 }
225
226 pub fn report_success(tree: &AbstractSyntaxTree) {
227     log::warn!(
228         "{} {}.",
229         "Transformed Abstract-Syntax Tree generation",
230         "success".bold().green()
231     );
232     log::info!("{}", tree.pretty_print());
233 }
234
235 pub fn report_error(error: AstError) {
236     log::error!(
237         "{} {}",
238         "Transformed Abstract-Syntax Tree generation:".bold().red(),
239         error

```

240);
241 }
242

balancer.rs

```
1  use crate::compiler::ast::tree::{
2      AbstractSyntaxTree, AstError, AstNode, BinaryOperationKind,
3  };
4  use colored::Colorize;
5  use std::collections::VecDeque;
6
7  impl AbstractSyntaxTree {
8      pub fn balance(self) -> Result<Self, AstError> {
9          let peek = Self::balance_tree(self.peek)?;
10
11          Ok(Self::from_node(peek))
12      }
13
14      pub fn balance_tree(node: AstNode) -> Result<AstNode, AstError> {
15          match node {
16              // Base cases, already balanced.
17              AstNode::Number(_) | AstNode::Identifier(_) | AstNode::StringLiteral(_) => {
18                  Ok(node)
19              },
20
21              // Recursive cases for other node types.
22              AstNode::UnaryOperation {
23                  operation,
24                  expression,
25              } => Ok(AstNode::UnaryOperation {
26                  operation,
27                  expression: Box::new(Self::balance_tree(*expression)?),
28              }),
29
30              AstNode::FunctionCall { name, arguments } => {
31                  let mut balanced_arguments: Vec<AstNode> = vec![];
32                  for arg in arguments {
33                      balanced_arguments.push(Self::balance_tree(arg)?);
34                  }
35
36                  Ok(AstNode::FunctionCall {
37                      name,
38                      arguments: balanced_arguments,
39                  })
40              },
41
42              AstNode::ArrayAccess {
43                  identifier,
44                  indices,
45              } => {
```

```

46     let mut balanced_indices: Vec<AstNode> = vec![];
47     for index in indices {
48         balanced_indices.push(Self::balance_tree(index)?);
49     }
50
51     Ok(AstNode::ArrayAccess {
52         identifier,
53         indices: balanced_indices,
54     })
55 },
56
57 // Main logic: Binary operations
58 AstNode::BinaryOperation {
59     operation,
60     left,
61     right,
62 } => {
63     match operation {
64         BinaryOperationKind::Plus | BinaryOperationKind::Multiply => {
65             let mut operands = Vec::new();
66             Self::collect_operands(
67                 AstNode::BinaryOperation {
68                     operation: operation.clone(),
69                     left,
70                     right,
71                 },
72                 operation.clone(),
73                 &mut operands,
74             );
75
76             let mut balanced_operands = Vec::new();
77             for operand in operands {
78                 balanced_operands.push(Self::balance_tree(operand)?);
79             }
80
81             Self::build_balanced_tree(balanced_operands, operation)
82         },
83
84         // Other operations (And, Or, etc.) are not associative
85         // in the arithmetic context. Just return them
86         // with already balanced children.
87         _ => {
88             let balanced_left = Self::balance_tree(*left)?;
89             let balanced_right = Self::balance_tree(*right)?;
90             Ok(AstNode::BinaryOperation {
91                 operation,
92                 left: Box::new(balanced_left),
93                 right: Box::new(balanced_right),
94             })
95         },

```

```

96     }
97     },
98     }
99 }
100
101 /// Making flatten tree.
102 /// Recursively "unfolds" a chain of associative operations
103 /// into a flat list. For example, the tree '(a + (b + c)) + d'
104 /// with 'op_kind = Plus' will be "flattened" into the list '[a, b, c, d]'.
105 fn collect_operands(
106     node: AstNode, op_kind: BinaryOperationKind, operands: &mut Vec<AstNode>,
107 ) {
108     match node {
109         // If operation node is the same as we are looking for...
110         AstNode::BinaryOperation {
111             operation,
112             left,
113             right,
114         } if operation == op_kind => {
115             // ... we recursively collect operands from both sides.
116             Self::collect_operands(*left, op_kind.clone(), operands);
117             Self::collect_operands(*right, op_kind.clone(), operands);
118         },
119         // If operation node is different, or it's a leaf node...
120         // or just operand (a, b, c...),
121         // then it is a "leaf" for *this* chain.
122         // We add it to the list.
123         _ => {
124             operands.push(node);
125         },
126     }
127 }
128
129 /// Building balanced tree
130 /// Taking a flat list of operands and constructing
131 /// a binary tree of minimal height using a queue-based algorithm.
132 /// For example, '[a, b, c, d, e]' becomes '((a + b) + (c + d)) + e'
133 /// (or a similar balanced structure).
134 fn build_balanced_tree(
135     operands: Vec<AstNode>, op_kind: BinaryOperationKind,
136 ) -> Result<AstNode, AstError> {
137     if operands.is_empty() {
138         return Err(AstError::CannotBuildEmptyTree);
139     }
140
141     // Making a queue from the list of operands
142     let mut queue: VecDeque<AstNode> = operands.into();
143
144     // While more than one node remains in the queue...
145     while queue.len() > 1 {

```

```

146     let level_size = queue.len();
147
148     // Process the current level of the tree:
149     for _ in 0..(level_size / 2) {
150         // Take two nodes from the front of the queue...
151         let left = queue.pop_front().ok_or(AstError::FailedPopFromQueue)?;
152         let right = queue.pop_front().ok_or(AstError::FailedPopFromQueue)?;
153
154         // ...create a new binary operation node combining them...
155         let new_node = AstNode::BinaryOperation {
156             operation: op_kind.clone(),
157             left: Box::new(left),
158             right: Box::new(right),
159         };
160
161         // .. and put the new node at the back of the queue
162         // (it will be an operand for the next, higher level)
163         queue.push_back(new_node);
164     }
165
166     // if level_size is odd...
167     if !level_size.is_multiple_of(2) {
168         // ...one node remains at the front of the queue.
169         // We simply move it to the back,
170         // so it can participate in the next iteration (next level).
171         let odd_one_out =
172             queue.pop_front().ok_or(AstError::FailedPopFromQueue)?;
173         queue.push_back(odd_one_out);
174     }
175 }
176
177 // When only one node remains in the queue,
178 // it is the root of the balanced tree.
179 queue.pop_front().ok_or(AstError::FailedPopFromQueue)
180 }
181 }
182
183 pub fn report_success(tree: &AbstractSyntaxTree) {
184     log::warn!(
185         "{} {}. ",
186         "Balanced Abstract-Syntax Tree generation",
187         "success".bold().green()
188     );
189     log::info!("{}", tree.pretty_print());
190 }
191
192 pub fn report_error(error: AstError) {
193     log::error!("{}", "Balanced AST error:".bold().red(), error);
194 }
195

```