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

КИЇВ – 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.

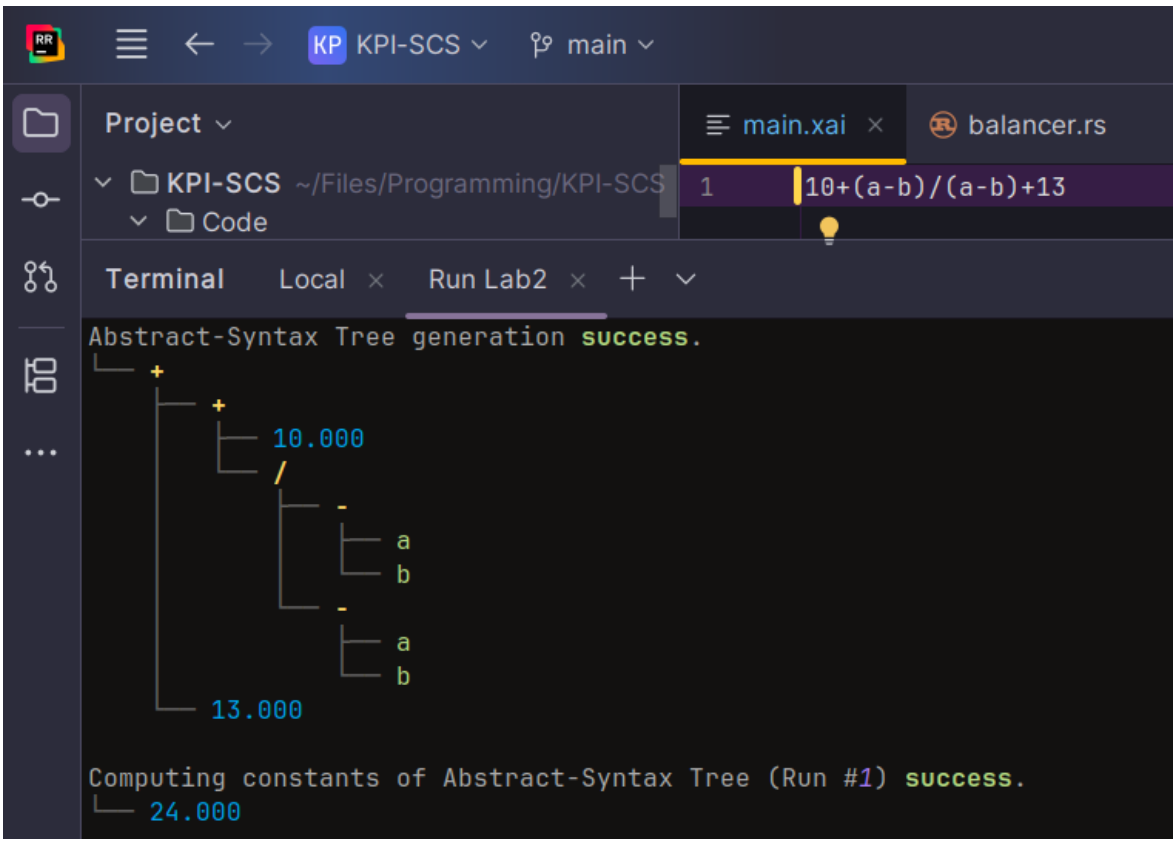
Основою для всіх подальших трансформацій є абстрактне синтаксичне дерево (AST). Спочатку береться синтаксично коректний потік лексем (сформований `lexer.rs`) і перетворює його на деревовидну структуру. Для цього використовується `AstParser` – парсер, реалізований за методом рекурсивного спуску. Він послідовно 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.

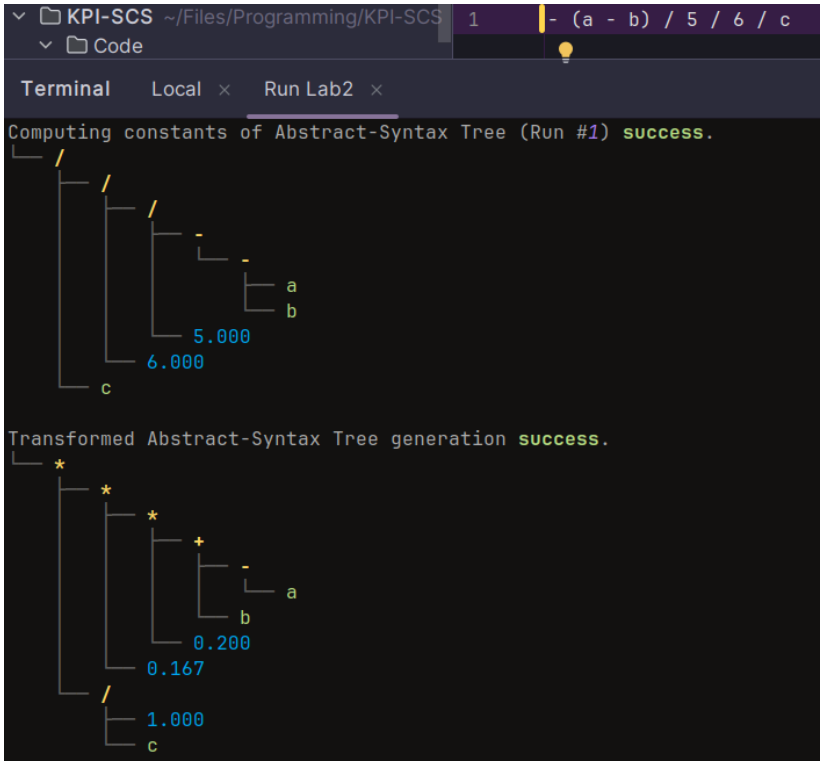
Демонстрація згортання констант:



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) -> 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.

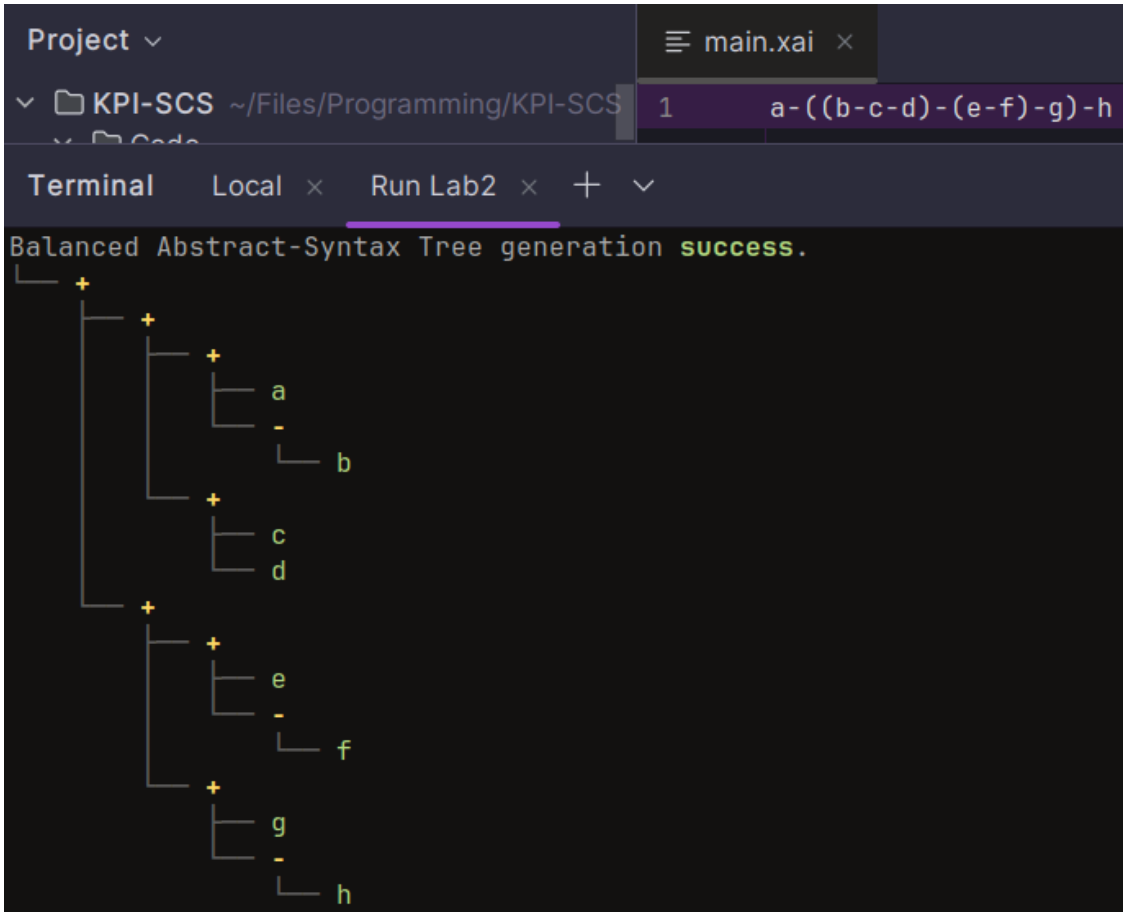
Демонстрація операції трансформації:



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, спрощую чи `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

```rust
use crate::compiler::ast::tree::{AbstractSyntaxTree, AstParser};
use crate::compiler::lexer::Lexer;
use crate::compiler::syntax::SyntaxAnalyzer;

pub fn compile(source: &str, is_pretty: bool) {
  // Lexical Analysis
  let tokens = tokenizer::tokenize(source);
  // Syntax Analysis
  let syntax_errors = SyntaxAnalyzer::new(&tokens).analyze();
  let is_syntax_analysis_successful = syntax_errors.is_empty();
  syntax::report(source, syntax_errors, is_pretty);
  if !is_syntax_analysis_successful {
    return;
  }

  // Making lexemes
  let lexemes_result = Lexer::new(tokens).run();
  let lexemes = match lexemes_result {
    Ok(lexemes) => {
      lexer::report_success(&lexemes);
      lexemes
    },
    Err(error) => {
      lexer::report_error(error);
      return;
    },
  };

  // AST Generation
  let ast_result = AstParser::new(lexemes).parse();
  let ast = match ast_result {
    Ok(ast) => {
      ast::tree::report_success(&ast);
      ast
    },
    Err(error) => {
      ast::tree::report_error(error);
      return;
    },
  };
  // AST Math Optimization, #1
  let ast = match compute_run(ast, 1) {
    Some(ast) => ast,
    None => return,
  };
  // AST Parallelization
  let ast_result = ast.transform();
```

```
    let ast = match ast_result {
      Ok(ast) => {
        ast::transform::report_success(&ast);
        ast
      },
      Err(error) => {
        ast::transform::report_error(error);
        return;
      },
    };
    // AST Math Optimization, #2
    let ast = match compute_run(ast, 2) {
      Some(ast) => ast,
      None => return,
    };
    // AST Balancing
    let ast_result = ast.balance();
    let ast = match ast_result {
      Ok(ast) => {
        ast::balancer::report_success(&ast);
        ast
      },
      Err(error) => {
        ast::balancer::report_error(error);
        return;
      },
    };
    // AST Math Optimization, #3
    let _ast = match compute_run(ast, 3) {
      Some(ast) => ast,
      None => return,
    };
  }

  fn compute_run(tree: AbstractSyntaxTree, number: u8) -> Option<AbstractSyntaxTree> {
    // AST Math Optimization
    let ast_result = tree.compute();
    let ast = match ast_result {
      Ok(ast) => {
        ast::math::report_success(&ast, number);
        ast
      },
      Err(error) => {
        ast::math::report_error(error, number);
        return None;
      },
    };
    if ast::math::check_finalization(&ast) {
      return None;
    }
```

```
    Some(ast)
  }

  pub mod ast {
    pub mod balancer;
    pub mod math;
    pub mod transform;
    pub mod tree;
  }
  pub mod lexer;
  pub mod syntax;
  pub mod tokenizer;

```

tree.rs

```
use crate::compiler::lexer::Lexeme;
use colored::Colorize;

#[derive(Debug, Clone, PartialEq)]
pub struct AbstractSyntaxTree {
  pub peek: AstNode,
}

impl AbstractSyntaxTree {
  pub fn from_node(node: AstNode) -> Self {
    Self { peek: node }
  }

  pub fn pretty_print(&self) -> String {
    let mut tree = String::new();
    Self::print_recursive(&self.peek, &mut tree, "".to_string(), true);
    tree
  }

  fn print_recursive(node: &AstNode, tree: &mut String, prefix: String, is_last: bool) {
    let connector = if is_last { "└── " } else { "├── " };

    tree.push_str(&format!("{}{}", prefix.dimmed(), connector.dimmed()));

    let node_text = match node {
      AstNode::Number(n) => format!("{n:.3}").bright_blue(),
      AstNode::Identifier(s) => s.to_string().green(),
      AstNode::StringLiteral(s) => format!("\"{}\"", s).bright_magenta(),
      AstNode::UnaryOperation { operation, .. } => {
        operation.to_string().yellow().bold()
      },
      AstNode::BinaryOperation { operation, .. } => {
        operation.to_string().yellow().bold()
      },
      AstNode::FunctionCall { name, .. } => format!("{}(...)", name).cyan().bold(),
```

```
        AstNode::ArrayAccess { identifier, .. } => {
          format!("{}[...]", identifier).blue().bold()
        },
      };
      tree.push_str(&format!("{}\n", node_text));

      let new_prefix = prefix + if is_last { "    " } else { "|   " };

      match node {
        AstNode::Number(_) | AstNode::Identifier(_) | AstNode::StringLiteral(_) => {},

        AstNode::UnaryOperation { expression, .. } => {
          Self::print_recursive(expression, tree, new_prefix, true);
        },

        AstNode::BinaryOperation { left, right, .. } => {
          Self::print_recursive(left, tree, new_prefix.clone(), false);
          Self::print_recursive(right, tree, new_prefix, true);
        },

        AstNode::FunctionCall { arguments, .. } => {
          let arg_count = arguments.len();
          for (i, arg) in arguments.iter().enumerate() {
            let is_last_arg = i == arg_count - 1;
            Self::print_recursive(arg, tree, new_prefix.clone(), is_last_arg);
          }
        },

        AstNode::ArrayAccess {
          identifier: _,
          indices,
        } => {
          let dimensions = indices.len();
          for (i, index) in indices.iter().enumerate() {
            let is_last_arg = i == dimensions - 1;
            Self::print_recursive(index, tree, new_prefix.clone(), is_last_arg);
          }
        },
      }
    }
  }

  #[derive(Debug, Clone, PartialEq)]
  pub enum AstNode {
    Number(f64),
    Identifier(String),
    StringLiteral(String),
    UnaryOperation {
      operation: UnaryOperationKind,
      expression: Box<AstNode>,
```

```
    },
    BinaryOperation {
      operation: BinaryOperationKind,
      left: Box<AstNode>,
      right: Box<AstNode>,
    },
    FunctionCall {
      name: String,
      arguments: Vec<AstNode>,
    },
    ArrayAccess {
      identifier: String,
      indices: Vec<AstNode>,
    },
  }

  #[derive(Debug, Clone, PartialEq)]
  pub enum UnaryOperationKind {
    Minus,
    Not,
  }

  #[derive(Debug, Clone, PartialEq)]
  pub enum BinaryOperationKind {
    Plus,
    Minus,
    Multiply,
    Divide,
    Or,
    And,
  }

  pub struct AstParser {
    lexemes: Vec<Lexeme>,
    current_index: usize,
  }

  impl AstParser {
    pub fn new(lexemes: Vec<Lexeme>) -> Self {
      Self {
        lexemes,
        current_index: 0,
      }
    }

    pub fn parse(&mut self) -> Result<AbstractSyntaxTree, AstError> {
      let node = self.parse_logical_or()?;

      if self.peek().is_some()
      && let Some(peek) = self.consume()
```

```
        {
          Err(AstError::NotExpectedLexeme(peek.clone()))
        } else {
          Ok(AbstractSyntaxTree { peek: node })
        }
      }

      fn parse_logical_or(&mut self) -> Result<AstNode, AstError> {
        let mut left_node = self.parse_logical_and()?;

        while let Some(Lexeme::Or) = self.peek()
        && let Some(_) = self.consume()
        {
          let right_node = self.parse_logical_and()?;
          left_node = AstNode::BinaryOperation {
            operation: BinaryOperationKind::Or,
            left: Box::new(left_node),
            right: Box::new(right_node),
          };
        }
        Ok(left_node)
      }

      fn parse_logical_and(&mut self) -> Result<AstNode, AstError> {
        let mut left_node = self.parse_expression()?;

        while let Some(Lexeme::And) = self.peek()
        && let Some(_) = self.consume()
        {
          let right_node = self.parse_expression()?;
          left_node = AstNode::BinaryOperation {
            operation: BinaryOperationKind::And,
            left: Box::new(left_node),
            right: Box::new(right_node),
          };
        }
        Ok(left_node)
      }

      fn parse_expression(&mut self) -> Result<AstNode, AstError> {
        let mut left_node = self.parse_term()?;

        while let Some(Lexeme::Plus) | Some(Lexeme::Minus) = self.peek()
        && let Some(lexeme) = self.consume()
        {
          let operation = match lexeme {
            Lexeme::Plus => BinaryOperationKind::Plus,
            Lexeme::Minus => BinaryOperationKind::Minus,
            _ => return Err(AstError::UnreachableLexeme(lexeme.clone())),
          };
```

```
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 {
```

```
            operation: operation_kind,
            expression: Box::new(child_node),
          })
        } else {
          self.parse_primary()
        }
      }

      fn parse_primary(&mut self) -> Result<AstNode, AstError> {
        if let Some(lexeme) = self.consume() {
          match lexeme {
            Lexeme::Number(value) => Ok(AstNode::Number(value)),
            Lexeme::String(value) => {
              match (matches!(self.peek(), Some(Lexeme::Comma)))
              || (matches!(self.peek_previous_by(2), Some(Lexeme::Comma)))
              {
                true => Ok(AstNode::StringLiteral(value.clone())),
                false => Err(AstError::StringOutsideFunction(value.clone())),
              }
            },

            Lexeme::LeftParenthesis => {
              let inner_node = self.parse_logical_or()?;

              if self.peek() == Some(&Lexeme::RightParenthesis) {
                self.consume();
                Ok(inner_node)
              } else {
                Err(AstError::ExpectedRightParenthesis)
              }
            },

            Lexeme::Identifier(name) => {
              if self.peek() == Some(&Lexeme::LeftParenthesis)
              && let Some(_) = self.consume()
              {
                let function_name = name.clone();
                let mut args = Vec::new();

                if self.peek() != Some(&Lexeme::RightParenthesis) {
                  loop {
                    args.push(self.parse_logical_or()?);

                    let peek = self.peek();

                    if peek == Some(&Lexeme::Comma) {
                      let _ = self.consume();
                    } else if peek == Some(&Lexeme::RightParenthesis) {
                      break;
                    } else {
```

```
                    return Err(match peek {
                      None => AstError::NotExpectedEndOfExpression,
                      Some(lexeme) => {
                        AstError::ExpectedCommaOrRightParenthesis(
                        lexeme.clone(),
                        )
                      },
                    });
                  }
                }
              }

              let _ = self.consume();

              Ok(AstNode::FunctionCall {
                name: function_name,
                arguments: args,
              })
            } else if self.peek() == Some(&Lexeme::LeftBracket) {
              let identifier = name.clone();
              let mut indices: Vec<AstNode> = Vec::new();

              loop {
                let _ = self.consume();
                let index = self.parse_logical_or()?;
                if self.peek() == Some(&Lexeme::RightBracket) {
                  let _ = self.consume();
                  indices.push(index);
                  if self.peek() == Some(&Lexeme::LeftBracket) {
                    continue;
                  } else {
                    break;
                  }
                } else {
                  return Err(AstError::ExpectedRightBracket);
                }
              }
              Ok(AstNode::ArrayAccess {
                identifier,
                indices,
              })
            } else {
              Ok(AstNode::Identifier(name.clone()))
            }
          },

          _ => Err(AstError::NotExpectedLexeme(lexeme.clone())),
        }
      } else {
        Err(AstError::NotExpectedEndOfExpression)
```

```
      }
    }

    fn consume(&mut self) -> Option<Lexeme> {
      if let Some(lexeme) = self.peek() {
        let lexeme = lexeme.clone();
        self.current_index += 1;
        return Some(lexeme);
      }
      None
    }

    fn peek(&self) -> Option<&Lexeme> {
      self.lexemes.get(self.current_index)
    }

    fn peek_previous_by(&self, by: usize) -> Option<&Lexeme> {
      self.lexemes.get(self.current_index - by)
    }
  }

  pub fn report_success(tree: &AbstractSyntaxTree) {
    log::warn!(
    "{} {}.",
    "Abstract-Syntax Tree generation",
    "success".bold().green()
    );
    log::info!("{}", tree.pretty_print());
  }

  pub fn report_error(error: AstError) {
    log::error!("{} {}", "AST error:".bold().red(), error);
  }

  #[derive(Debug, PartialEq)]
  pub enum AstError {
    ExpectedRightBracket,
    ExpectedRightParenthesis,
    ExpectedCommaOrRightParenthesis(Lexeme),
    NotExpectedEndOfExpression,
    NotExpectedLexeme(Lexeme),
    StringOutsideFunction(String),
    UnreachableLexeme(Lexeme),

    CannotBuildEmptyTree,
    FailedPopFromQueue,
    DivisionByZero(AstNode),
  }

  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, "&"),
```

```
      }
    }
  }

```

math.rs

```rust
use crate::compiler::ast::tree::{
  AbstractSyntaxTree, AstError, AstNode, BinaryOperationKind, UnaryOperationKind,
};
use colored::Colorize;

impl AbstractSyntaxTree {
  pub fn compute(self) -> Result<AbstractSyntaxTree, AstError> {
    let computed = Self::compute_recursive(self.peek)?;

    Ok(Self::from_node(computed))
  }

  fn compute_recursive(node: AstNode) -> Result<AstNode, AstError> {
    match &node {
      AstNode::Number(_) | AstNode::Identifier(_) | AstNode::StringLiteral(_) => {
        Ok(node)
      },
      AstNode::UnaryOperation {
        operation: op,
        expression,
      } => match &op {
        UnaryOperationKind::Minus => {
          let child = Self::compute_recursive(*expression.clone())?;
          if let AstNode::Number(number) = child {
            Ok(AstNode::Number(-number))
          } else {
            Ok(node)
          }
        },
        UnaryOperationKind::Not => Ok(node),
      },
      AstNode::BinaryOperation {
        operation,
        left,
        right,
      } => match operation {
        BinaryOperationKind::Plus
        | BinaryOperationKind::Minus
        | BinaryOperationKind::Multiply
        | BinaryOperationKind::Divide => {
          let computed_left = Self::compute_recursive(*left.clone())?;
          let computed_right = Self::compute_recursive(*right.clone())?;

          // Case: (a + b) - (a + b) = 0
```

```
            // Or: (a + b) / (a + b) = 1
            if computed_left.eq(&computed_right) {
              match operation {
                BinaryOperationKind::Minus => {
                  return Ok(AstNode::Number(0.0));
                },
                BinaryOperationKind::Divide => {
                  if let AstNode::Number(number) = &computed_left
                  && *number == 0.0
                  {
                    // Case: (5 - 5) / (5 - 5)
                    return Err(AstError::DivisionByZero(node));
                  }
                  return Ok(AstNode::Number(1.0));
                },
                _ => {},
              }
            }

            if let (AstNode::Number(left_number), AstNode::Number(right_number)) =
            (&computed_left, &computed_right)
            {
              let result = match operation {
                BinaryOperationKind::Plus => left_number + right_number,
                BinaryOperationKind::Minus => left_number - right_number,
                BinaryOperationKind::Multiply => left_number * right_number,
                BinaryOperationKind::Divide => {
                  if *right_number == 0.0 {
                    return Err(AstError::DivisionByZero(node));
                  } else {
                    left_number / right_number
                  }
                },
                _ => unreachable!(),
              };
              Ok(AstNode::Number(result))
            } else if let AstNode::Number(number) = &computed_left {
              if number == &0.0 {
                if [
                BinaryOperationKind::Multiply,
                BinaryOperationKind::Divide,
                ]
                .contains(operation)
                {
                  return Ok(AstNode::Number(0.0));
                }
                if BinaryOperationKind::Plus == *operation {
                  return Ok(computed_right);
                }
                if BinaryOperationKind::Minus == *operation {
```

```
                    return Ok(AstNode::UnaryOperation {
                      operation: UnaryOperationKind::Minus,
                      expression: Box::new(computed_right),
                    });
                  }
                }
                if number == &1.0 && BinaryOperationKind::Multiply == *operation {
                  return Ok(computed_right);
                }

                Ok(AstNode::BinaryOperation {
                  operation: operation.clone(),
                  left: Box::new(computed_left),
                  right: Box::new(computed_right),
                })
              } else if let AstNode::Number(number) = &computed_right {
                if number == &0.0 {
                  if BinaryOperationKind::Divide == *operation {
                    return Err(AstError::DivisionByZero(node));
                  }
                  if BinaryOperationKind::Multiply == *operation {
                    return Ok(AstNode::Number(0.0));
                  }
                  if [BinaryOperationKind::Plus, BinaryOperationKind::Minus]
                  .contains(operation)
                  {
                    return Ok(computed_left);
                  }
                }
                if number == &1.0 && BinaryOperationKind::Multiply == *operation {
                  return Ok(computed_left);
                }

                Ok(AstNode::BinaryOperation {
                  operation: operation.clone(),
                  left: Box::new(computed_left),
                  right: Box::new(computed_right),
                })
              } else {
                Ok(AstNode::BinaryOperation {
                  operation: operation.clone(),
                  left: Box::new(computed_left),
                  right: Box::new(computed_right),
                })
              }
            },
            _ => Ok(node),
          },
          AstNode::FunctionCall { name, arguments } => {
            let mut computed_arguments = Vec::new();
```

```
            for arg in arguments {
              let arg = Self::compute_recursive(arg.clone())?;
              computed_arguments.push(arg);
            }

            Ok(AstNode::FunctionCall {
              name: name.clone(),
              arguments: computed_arguments,
            })
          },
          AstNode::ArrayAccess {
            identifier,
            indices,
          } => {
            let mut computed_indices = Vec::new();
            for index in indices {
              let index = Self::compute_recursive(index.clone())?;
              computed_indices.push(index);
            }
            Ok(AstNode::ArrayAccess {
              identifier: identifier.clone(),
              indices: computed_indices,
            })
          },
        }
      }
    }

    pub fn report_success(tree: &AbstractSyntaxTree, run: u8) {
      log::warn!(
      "Computing constants of Abstract-Syntax Tree (Run #{}) {}.",
      run.to_string().bright_magenta().italic(),
      "success".bold().green()
      );
      log::info!("{}", tree.pretty_print());
    }

    pub fn report_error(error: AstError, run: u8) {
      log::error!(
      "{} (Run #{}) {}",
      "Computing constants of Abstract-Syntax Tree:".bold().red(),
      run.to_string().bright_magenta().italic(),
      error
      );
    }

    pub fn check_finalization(tree: &AbstractSyntaxTree) -> bool {
      if let AstNode::Number(number) = &tree.peek {
        log::warn!(
        "{} = {}. {}.",
```

```
      "Computing solved code, result".bold().green(),
      number,
      "Further optimization is not needed".bold().red(),
      );
      return true;
    }
    false
  }

```

transform.rs

```
use crate::compiler::ast::tree::{
  AbstractSyntaxTree, AstError, AstNode, BinaryOperationKind, UnaryOperationKind,
};
use colored::Colorize;

impl AbstractSyntaxTree {
  pub fn transform(self) -> Result<AbstractSyntaxTree, AstError> {
    let peek = Self::transform_recursive(self.peek)?;

    Ok(Self::from_node(peek))
  }

  pub fn transform_recursive(node: AstNode) -> Result<AstNode, AstError> {
    match &node {
      AstNode::Number(_) | AstNode::Identifier(_) | AstNode::StringLiteral(_) => {
        Ok(node)
      },

      AstNode::UnaryOperation {
        operation,
        expression,
      } => {
        let transformed_expression =
        Self::transform_recursive(*expression.clone())?;
        match operation {
          UnaryOperationKind::Not => Ok(AstNode::UnaryOperation {
            operation: operation.clone(),
            expression: Box::new(transformed_expression),
          }),

          UnaryOperationKind::Minus => {
            match &transformed_expression {
              // Rule: -(-A) => A
              AstNode::UnaryOperation {
                operation: UnaryOperationKind::Minus,
                expression: inner_expr,
              } => Ok(*inner_expr.clone()),

              // Rule: -(A + B) => (-A) + (-B)
```

```
                AstNode::BinaryOperation {
                  operation: BinaryOperationKind::Plus,
                  left,
                  right,
                } => {
                  let new_left = AstNode::UnaryOperation {
                    operation: UnaryOperationKind::Minus,
                    expression: left.clone(),
                  };
                  let new_right = AstNode::UnaryOperation {
                    operation: UnaryOperationKind::Minus,
                    expression: right.clone(),
                  };
                  Self::transform_recursive(AstNode::BinaryOperation {
                    operation: BinaryOperationKind::Plus,
                    left: Box::new(new_left),
                    right: Box::new(new_right),
                  })
                },

                // Rule: -(A - B) => (-A) + B
                AstNode::BinaryOperation {
                  operation: BinaryOperationKind::Minus,
                  left,
                  right,
                } => {
                  let new_left = AstNode::UnaryOperation {
                    operation: UnaryOperationKind::Minus,
                    expression: left.clone(),
                  };
                  Self::transform_recursive(AstNode::BinaryOperation {
                    operation: BinaryOperationKind::Plus,
                    left: Box::new(new_left),
                    right: right.clone(),
                  })
                },

                // Rule: -(Num * B) => -Num * B
                AstNode::BinaryOperation {
                  operation: BinaryOperationKind::Multiply,
                  left,
                  right,
                } => {
                  if let AstNode::Number(number) = *left.clone() {
                    return Ok(AstNode::BinaryOperation {
                      operation: BinaryOperationKind::Multiply,
                      left: Box::new(AstNode::Number(-number)),
                      right: Box::new(*right.clone()),
                    });
                  }
```

```
                Ok(AstNode::UnaryOperation {
                  operation: UnaryOperationKind::Minus,
                  expression: Box::new(transformed_expression),
                })
              },

              // Other cases (for example, -(A*B) or just -A)
              // just leaving them (for example -(A*B) or just -A)
              _ => Ok(AstNode::UnaryOperation {
                operation: UnaryOperationKind::Minus,
                expression: Box::new(transformed_expression),
              }),
            }
          },
        }
      },

      AstNode::FunctionCall { name, arguments } => {
        let mut transformed_arguments = vec![];
        for argument in arguments {
          transformed_arguments
          .push(Self::transform_recursive(argument.clone())?);
        }

        Ok(AstNode::FunctionCall {
          name: name.clone(),
          arguments: transformed_arguments,
        })
      },

      AstNode::BinaryOperation {
        operation,
        left,
        right,
      } => {
        let transformed_left = Self::transform_recursive(*left.clone())?;
        let transformed_right = Self::transform_recursive(*right.clone())?;

        match operation {
          // Rule 1: A - B  =>  A + (-B)
          BinaryOperationKind::Minus => {
            match &transformed_right {
              AstNode::Number(number) if number.is_sign_negative() => {
                // Rule 1: A - (-B)  =>  A + B
                Ok(AstNode::BinaryOperation {
                  operation: BinaryOperationKind::Plus,
                  left: Box::new(transformed_left),
                  right: Box::new(AstNode::Number(f64::abs(*number))),
                })
```

```
                },
                AstNode::Number(number) => {
                  // Rule 1: A - B  =>  A + (-B)
                  Ok(AstNode::BinaryOperation {
                    operation: BinaryOperationKind::Plus,
                    left: Box::new(transformed_left),
                    right: Box::new(AstNode::Number(-number)),
                  })
                },
                _ => {
                  let new_right = AstNode::UnaryOperation {
                    operation: UnaryOperationKind::Minus,
                    expression: Box::new(transformed_right),
                  };

                  let result_node = AstNode::BinaryOperation {
                    operation: BinaryOperationKind::Plus,
                    left: Box::new(transformed_left),
                    right: Box::new(new_right),
                  };
                  Self::transform_recursive(result_node)
                },
              }
            },

            // Rule 2: A / B  =>  A * (1 / B)
            BinaryOperationKind::Divide => {
              if let AstNode::Number(number) = transformed_left
              && number == 1.0
              {
                return Ok(AstNode::BinaryOperation {
                  operation: BinaryOperationKind::Divide,
                  left: Box::new(transformed_left),
                  right: Box::new(transformed_right),
                });
              }

              if let AstNode::Number(number) = transformed_right {
                return if number == 0.0 {
                  Err(AstError::DivisionByZero(node))
                } else {
                  Ok(AstNode::BinaryOperation {
                    operation: BinaryOperationKind::Multiply,
                    left: Box::new(transformed_left),
                    right: Box::new(AstNode::Number(1.0 / number)),
                  })
                };
              }
              Ok(AstNode::BinaryOperation {
                operation: BinaryOperationKind::Multiply,
```

```
                left: Box::new(transformed_left),
                right: Box::new(AstNode::BinaryOperation {
                  operation: BinaryOperationKind::Divide,
                  left: Box::new(AstNode::Number(1.0)), // "1"
                  right: Box::new(transformed_right),   // "B"
                }),
              })
            },

            // Other operations (Plus, Multiply, And, Or)
            // left without editing, but with transformed kids.
            _ => Ok(AstNode::BinaryOperation {
              operation: operation.clone(),
              left: Box::new(transformed_left),
              right: Box::new(transformed_right),
            }),
          }
        },

        AstNode::ArrayAccess {
          identifier,
          indices,
        } => {
          let mut transformed_indices = vec![];
          for index in indices {
            transformed_indices.push(Self::transform_recursive(index.clone())?);
          }
          Ok(AstNode::ArrayAccess {
            identifier: identifier.clone(),
            indices: transformed_indices,
          })
        },
      }
    }
  }

  pub fn report_success(tree: &AbstractSyntaxTree) {
    log::warn!(
    "{} {}.",
    "Transformed Abstract-Syntax Tree generation",
    "success".bold().green()
    );
    log::info!("{}", tree.pretty_print());
  }

  pub fn report_error(error: AstError) {
    log::error!(
    "{} {}",
    "Transformed Abstract-Syntax Tree generation:".bold().red(),
    error
```

```
    );
  }

```

balancer.rs

```rust
use crate::compiler::ast::tree::{
  AbstractSyntaxTree, AstError, AstNode, BinaryOperationKind,
};
use colored::Colorize;
use std::collections::VecDeque;

impl AbstractSyntaxTree {
  pub fn balance(self) -> Result<Self, AstError> {
    let peek = Self::balance_tree(self.peek)?;

    Ok(Self::from_node(peek))
  }

  pub fn balance_tree(node: AstNode) -> Result<AstNode, AstError> {
    match node {
      // Base cases, already balanced.
      AstNode::Number(_) | AstNode::Identifier(_) | AstNode::StringLiteral(_) => {
        Ok(node)
      },

      // Recursive cases for other node types.
      AstNode::UnaryOperation {
        operation,
        expression,
      } => Ok(AstNode::UnaryOperation {
        operation,
        expression: Box::new(Self::balance_tree(*expression)?),
      }),

      AstNode::FunctionCall { name, arguments } => {
        let mut balanced_arguments: Vec<AstNode> = vec![];
        for arg in arguments {
          balanced_arguments.push(Self::balance_tree(arg)?);
        }

        Ok(AstNode::FunctionCall {
          name,
          arguments: balanced_arguments,
        })
      },

      AstNode::ArrayAccess {
        identifier,
        indices,
      } => {
```

```
            let mut balanced_indices: Vec<AstNode> = vec![];
            for index in indices {
              balanced_indices.push(Self::balance_tree(index)?);
            }

            Ok(AstNode::ArrayAccess {
              identifier,
              indices: balanced_indices,
            })
          },

          // Main logic: Binary operations
          AstNode::BinaryOperation {
            operation,
            left,
            right,
          } => {
            match operation {
              BinaryOperationKind::Plus | BinaryOperationKind::Multiply => {
                let mut operands = Vec::new();
                Self::collect_operands(
                AstNode::BinaryOperation {
                  operation: operation.clone(),
                  left,
                  right,
                },
                operation.clone(),
                &mut operands,
                );

                let mut balanced_operands = Vec::new();
                for operand in operands {
                  balanced_operands.push(Self::balance_tree(operand)?);
                }

                Self::build_balanced_tree(balanced_operands, operation)
              },

              // Other operations (And, Or, etc.) are not associative
              // in the arithmetic context. Just return them
              // with already balanced children.
              _ => {
                let balanced_left = Self::balance_tree(*left)?;
                let balanced_right = Self::balance_tree(*right)?;
                Ok(AstNode::BinaryOperation {
                  operation,
                  left: Box::new(balanced_left),
                  right: Box::new(balanced_right),
                })
              },
```

```
            }
          },
        }
      }

      /// Making flatten tree.
      /// Recursively "unfolds" a chain of associative operations
      /// into a flat list. For example, the tree `(a + (b + c)) + d`
      /// with `op_kind = Plus` will be "flattened" into the list `[a, b, c, d]`.
      fn collect_operands(
      node: AstNode, op_kind: BinaryOperationKind, operands: &mut Vec<AstNode>,
      ) {
        match node {
          // If operation node is the same as we are looking for...
          AstNode::BinaryOperation {
            operation,
            left,
            right,
          } if operation == op_kind => {
            // ... we recursively collect operands from both sides.
            Self::collect_operands(*left, op_kind.clone(), operands);
            Self::collect_operands(*right, op_kind.clone(), operands);
          },
          // If operation node is different, or it's a leaf node...
          // or just operand (a, b, c...),
          // then it is a "leaf" for *this* chain.
          // We add it to the list.
          _ => {
            operands.push(node);
          },
        }
      }

      /// Building balanced tree
      /// Taking a flat list of operands and constructing
      /// a binary tree of minimal height using a queue-based algorithm.
      /// For example, `[a, b, c, d, e]` becomes `((a + b) + (c + d)) + e`
      /// (or a similar balanced structure).
      fn build_balanced_tree(
      operands: Vec<AstNode>, op_kind: BinaryOperationKind,
      ) -> Result<AstNode, AstError> {
        if operands.is_empty() {
          return Err(AstError::CannotBuildEmptyTree);
        }

        // Making a queue from the list of operands
        let mut queue: VecDeque<AstNode> = operands.into();

        // While more than one node remains in the queue...
        while queue.len() > 1 {
```

```rust
        let level_size = queue.len();

        // Process the current level of the tree:
        for _ in 0..(level_size / 2) {
          // Take two nodes from the front of the queue...
          let left = queue.pop_front().ok_or(AstError::FailedPopFromQueue)?;
          let right = queue.pop_front().ok_or(AstError::FailedPopFromQueue)?;

          // ...create a new binary operation node combining them...
          let new_node = AstNode::BinaryOperation {
            operation: op_kind.clone(),
            left: Box::new(left),
            right: Box::new(right),
          };

          // .. and put the new node at the back of the queue
          // (it will be an operand for the next, higher level)
          queue.push_back(new_node);
        }

        // if level_size is odd...
        if !level_size.is_multiple_of(2) {
          // ...one node remains at the front of the queue.
          // We simply move it to the back,
          // so it can participate in the next iteration (next level).
          let odd_one_out =
          queue.pop_front().ok_or(AstError::FailedPopFromQueue)?;
          queue.push_back(odd_one_out);
        }
      }

      // When only one node remains in the queue,
      // it is the root of the balanced tree.
      queue.pop_front().ok_or(AstError::FailedPopFromQueue)
    }
  }

  pub fn report_success(tree: &AbstractSyntaxTree) {
    log::warn!(
    "{} {}.",
    "Balanced Abstract-Syntax Tree generation",
    "success".bold().green()
    );
    log::info!("{}", tree.pretty_print());
  }

  pub fn report_error(error: AstError) {
    log::error!("{} {}", "Balanced AST error:".bold().red(), error);
  }

```