

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

Факультет інформатики та обчислювальної техніки  
Кафедра обчислювальної техніки

## ЗВІТ

з лабораторних робіт №3-4  
з дисципліни ”Програмне забезпечення комп’ютерних систем”

**Тема: Знаходження еквівалентних форм АВ**

**Варіант №5**

Виконав:

Студент 1 курсу, групи ІМ-51мн  
Ковальов Олександр

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

Дата здачі: 11.11.2025

КИЇВ – 2025

**Мета роботи.** Визначення множини еквівалентних форм АВ з використанням заданих алгебраїчних законів.

**Вхідні дані:** Коректний арифметичний вираз в аналітичному вигляді після успішного виконання лексичного та синтаксичного аналізу (результат виконання лабораторної роботи №1) або дерево паралельної форми арифметичного виразу (результат виконання лабораторної роботи №2). Крім того, кожний студент отримує два типи перетворень, які необхідно реалізувати відповідно у лабораторних роботах №3 та №4.

**Завдання:** У лабораторних роботах №3 та №4 на основі перерахованих вхідних даних розробити алгоритми та програмно реалізувати отримання множини еквівалентних форм АВ або відповідних дерев паралельних форм з урахуванням алгебраїчних законів комутативності, дистрибутивності, асоціативності, а також з розкриття дужок.

### **Хід роботи.**

В рамках виконання лабораторної роботи №3-4 було реалізовано механізм пошуку еквівалентних форм арифметичного виразу. Завдання полягало у застосуванні дистрибутивного (розкриття дужок) та асоціативного (винесення спільного множника) законів для генерації множини всіх унікальних форм, що походять від вхідного виразу.

За основу стратегії пошуку було взято двоетапний алгоритм, що відповідає графу, наведеному у методичних матеріалах. Цей підхід чітко розділяє процес на дві фази: розширення та факторинг. Для керування пошуком та уникнення дублікатів і циклів використовується структура даних HashSet, яка зберігає канонічні рядкові представлення кожної вже відвіданої форми. Основний алгоритм пошуку реалізований як пошук у ширину (BFS) з використанням черги VecDeque для кожного етапу, що гарантує покрове знаходження форм.

Перший етап, розширення, починається з оригінального виразу. На кожній ітерації BFS викликається функція get\_all\_single\_step\_expansions. Ця функція рекурсивно сканує поточне дерево AST у пошуках вузлів BinaryOperationKind::Multiply, у яких один з операндів є бінарною операцією Plus або Minus. Це відповідає патернам  $A * (B +/ - C)$  або  $(A +/ - B) * C$ . Для кожного знайденого вузла створюється нова копія дерева, де цей вузол замінюється на його розкритий еквівалент, наприклад,  $(A * B) + (A * C)$ . Нові згенеровані форми додаються до черги, якщо вони не були відвідані раніше. Цей процес триває, доки черга розширення не вичерпається. Кінцевою точкою цього етапу є

форма, в якій більше неможливо застосувати дистрибутивний закон (розкриття дужок множенням), що відповідає Вузлу 7 на графі з пам'ятки.

Важливим проміжним кроком, доданим між двома фазами, є "вирівнювання" (flattening) отриманої повністю розкритої форми. Форма, що є кінцевою для Етапу 1, може все ще містити дужки, наприклад,  $\dots - (a * x - t * x)$ . Щоб отримати істинно "пласку" форму без дужок, до цього вузла застосовуються функції `transform_recursive` та `fold_recursive`. Функція `transform_recursive` рекурсивно застосовує правила перетворення унарного мінуса, наприклад,  $- (A - B)$  на  $(-A) + B$ . Одразу після цього `fold_recursive` спрошує структуру, наприклад,  $A + (-B)$  на  $A - B$ . Результатом є нова, канонічна форма (у виводі це Форма 9), яка не містить зайвих дужок і слугує стартовою точкою для наступного етапу. Ця нова форма також додається до загального списку знайдених форм.

Другий етап, факторинг, починає свій пошук BFS саме з цієї нової "пласкої" форми. Логіка асоціативного закону є значно складнішою. Ключовий алгоритм `get_all_possible_factorings` спершу "вирівнює" дерево додавання/віднімання за допомогою функції `local_collect_operands`. Ця функція коректно обробляє операції `Minus`, перетворюючи вираз  $A - B$  на список доданків, де другий operand представлений як `UnaryMinus ((-B))`. Таким чином, вираз  $ak - ck - ax$  перетворюється на плаский список `[ (ak), (-ck), (-ax) ]`. Далі алгоритм ітерує по кожному доданку, знаходячи його множники за допомогою `get_factors` (яка коректно працює з унарним мінусом, наприклад, знаходячи  $a$  у  $-(a*x)$ ). Для кожного знайденого фактора (наприклад,  $a$ ), він шукає цей фактор в інших доданках списку. Залишки збираються за допомогою `get_remainder` (яка також обробляє знаки, наприклад, `get_remainder(-(a*x), a)` повертає  $(-x)$ ). Якщо знайдено групу з кількох доданків зі спільним фактором, з їхніх залишків (наприклад, `[k, (-x)]`) будується нове збалансоване дерево додавання  $k + (-x)$ , яке потім множиться на спільний фактор  $a * (k + (-x))$ . Цей новий вузол разом з рештою доданків, що не увійшли до групи, формує нову еквівалентну форму, яка додається до черги BFS для подальшого аналізу.

Нарешті, для забезпечення читабельного виводу результатів, було реалізовано окрему функцію `to_pretty_string`. На відміну від `to_canonical_string`, яка використовується для `HashSet` і містить надлишкові дужки, "красивий" друк рекурсивно обходить дерево, враховуючи пріоритет операцій. Дужки додаються лише тоді, коли пріоритет дочірньої операції нижчий за батьківський (наприклад,  $(a + b) * c$ ). Крім того, до цієї функції було додано спеціальну логіку для коректного форматування унарних мінусів: конструкції

AST виду  $A + (-B)$  автоматично перетворюються на рядок " $A - B$ ", а  $(-A) + B$  – на " $B - A$ ". Це дозволило отримати фінальний список з 25 форм, включаючи повністю розкриту форму (Форма 9), у чистому та зрозумілому алгебраїчному вигляді.

```

Code: (a-c)*k + (b-c)*t - (a-t)*x
Run:
Tokenizer
Syntax check
Create Lexemes
Build AST
Compute AST #1
Transform AST
Compute AST #2
Balance AST
Compute AST #3
Fold AST
Compute AST #4
Equivalent Forms
Settings
Pretty Output
Save Config

```

Found 25 equivalent forms!

- 1)  $(a - c) * k + (b - c) * t - (a - t) * x$
- 2)  $a * k - c * k + (b - c) * t - (a - t) * x$
- 3)  $(a - c) * k + b * t - c * t - (a - t) * x$
- 4)  $(a - c) * k + (b - c) * t - (a * x - t * x)$
- 5)  $a * k - c * k + b * t - c * t - (a - t) * x$
- 6)  $a * k - c * k + (b - c) * t - (a * x - t * x)$
- 7)  $(a - c) * k + b * t - c * t - (a * x - t * x)$
- 8)  $a * k - c * k + b * t - c * t - (a * x - t * x)$
- 9)  $a * k - c * k + b * t - c * t + t * x - a * x$
- 10)  $a * (k - x) - c * k + b * t - c * t + t * x$
- 11)  $k * (a - c) + b * t + -(c * t) - a * x + t * x$
- 12)  $c * (-k - t) + a * k + b * t - a * x + t * x$
- 13)  $t * (b - c + x) + a * k + -(c * k) - a * x$
- 14)  $x * (t - a) + a * k + b * t - c * k - c * t$
- 15)  $c * (-k - t) + a * (k - x) + b * t + t * x$
- 16)  $t * (b - c + x) + a * (k - x) - c * k$
- 17)  $t * (b - c + x) + k * (a - c) - a * x$
- 18)  $x * (t - a) + k * (a - c) + b * t - c * t$
- 19)  $t * (b + x) + c * (-k - t) + a * k - a * x$
- 20)  $x * (t - a) + c * (-k - t) + a * k + b * t$
- 21)  $t * (b - c) + x * (t - a) + a * k - c * k$
- 22)  $t * (b + x) + c * (-k - t) + a * (k - x)$
- 23)  $t * (b - c) + x * (t - a) + k * (a - c)$
- 24)  $a * (k - x) + t * (b + x) + c * (-k - t)$
- 25)  $k * (a - c) + t * (b - c) + x * (t - a)$

**Висновок.** Під час виконання лабораторної роботи №3-4 було успішно розроблено та програмно реалізовано алгоритм для знаходження множини еквівалентних форм арифметичного виразу. Було імплементовано двоетапний підхід, що базується на графовій моделі з методичних матеріалів: спочатку застосовується дистрибутивний закон для повного розкриття дужок (розширення), а потім – асоціативний закон для винесення спільних множників (факторинг). В результаті роботи програма коректно генерує повний список унікальних, алгебраїчно правильних форм, включаючи повністю "плаский" вираз без дужок, що повністю відповідає меті лабораторної роботи.

## Лістинг.

### associative.rs

```
1  use crate::compiler::ast::tree::{
2      AbstractSyntaxTree, AstError, AstNode, BinaryOperationKind,
3  };
4  use std::collections::{HashSet, VecDeque};
5
6  impl AbstractSyntaxTree {
7      /// Returns a VECTOR of trees, where each tree is ONE step of the factoring.
8      pub fn get_all_single_step_factorings(&self) -> Vec<AbstractSyntaxTree> {
9          let mut all_new_forms = Vec::new();
10         // We start factoring from the root node.
11         Self::get_all_possible_factorings(self.peek.clone(), &mut all_new_forms);
12         all_new_forms
13     }
14
15     /// Finds *all* possible *single* factoring steps from the current node.
16     fn get_all_possible_factorings(
17         node: AstNode, all_forms: &mut Vec<AbstractSyntaxTree>,
18     ) {
19         // We can only factor terms from an addition or subtraction chain.
20         match &node {
21             AstNode::BinaryOperation { operation, .. }
22             if *operation == BinaryOperationKind::Plus
23             || *operation == BinaryOperationKind::Minus =>
24             {
25                 // This is a node we can start collecting from.
26             },
27             _ => return, // Not an additive/subtractive node, nothing to factor.
28         };
29
30         // --- STEP 1: Flatten the expression ---
31         // This is the CRITICAL FIX. We flatten the tree into a list of terms.
32         // 'ak - ck - ax' will become '[ (ak), (-ck), (-ax) ]'
33         let mut summands = Vec::new();
34         Self::local_collect_operands(node, &mut summands);
35
36         if summands.len() < 2 {
37             return; // Not enough terms to factor.
38         }
39
40         // --- STEP 2: Find all possible factor groupings ---
41         let mut unique_factors: HashSet<String> = HashSet::new();
42
43         // Iterate over every term to use as the "base" for a group
44         for i in 0..summands.len() {
45             let factors_i = Self::get_factors(&summands[i]);
46             if factors_i.is_empty() {
47                 continue; // This term (e.g., 'a') has no factors.
48             }
49             for j in (i + 1)..summands.len() {
50                 let factors_j = Self::get_factors(&summands[j]);
51                 if factors_j.is_empty() {
52                     continue; // This term (e.g., 'a') has no factors.
53                 }
54                 let mut factors_ij = factors_i.clone();
55                 factors_ij.extend(factors_j);
56                 unique_factors.insert(factors_ij.to_string());
57             }
58         }
59
60         all_forms.extend(unique_factors);
61     }
62 }
```

```

48 }
49
50 // Iterate over each factor of the base term (e.g., 'a' and 'k' for 'ak')
51 for factor in factors_i {
52     let tree = AbstractSyntaxTree::from_node(factor);
53     let factor_key = tree.to_canonical_string();
54     let factor = tree.peek;
55
56     // We use a HashSet to avoid generating duplicates.
57     if !unique_factors.insert(factor_key) {
58         continue;
59     }
60
61     // Get the remainder for the base term (e.g., 'k' from 'ak' with factor 'a')
62     if let Some(remainder_i) = Self::get_remainder(&summands[i], &factor) {
63         let mut current_group_terms = vec![remainder_i];
64         let mut current_group_indices = vec![i];
65
66         // --- STEP 3: Find other terms with the same factor ---
67         for (j, summand) in summands.iter().enumerate().skip(i + 1) {
68             // e.g., check '-ax' for factor 'a'
69             if let Some(remainder_j) = Self::get_remainder(summand, &factor) {
70                 // Remainder will be '(-x)'. This is correct.
71                 current_group_terms.push(remainder_j);
72                 current_group_indices.push(j);
73             }
74         }
75
76         // --- STEP 4: Build the new factored form ---
77         if current_group_terms.len() > 1 {
78             // We found a group! (e.g., factor 'a' with remainders [k, (-x)])
79             let mut new_summands = Vec::new();
80
81             // 1. Create the new factored node: a * (k + (-x))
82             // We *always* sum the remainders with 'Plus'.
83             let sum_of_terms = Self::local_build_balanced_tree(
84                 current_group_terms,
85                 BinaryOperationKind::Plus,
86             )
87             .unwrap_or(AstNode::Number(0.0)); // Should not happen
88
89             let new_node = AstNode::BinaryOperation {
90                 operation: BinaryOperationKind::Multiply,
91                 left: Box::new(factor.clone()),
92                 right: Box::new(sum_of_terms),
93             };
94             new_summands.push(new_node);
95
96             // 2. Add all the terms that were *not* part of this group
97             for (idx, term) in summands.iter().enumerate() {

```

```

98         if !current_group_indices.contains(&idx) {
99             new_summands.push(term.clone());
100        }
101    }
102
103    // 3. Re-assemble the final tree from all terms
104    // We *always* build the final tree with 'Plus' as well.
105    if let Ok(final_node) = Self::local_build_balanced_tree(
106        new_summands,
107        BinaryOperationKind::Plus,
108    ) {
109        all_forms.push(AbstractSyntaxTree::from_node(final_node));
110    }
111}
112}
113}
114}
115}
116
117 /// Flattens a chain of 'Plus'/'Minus' nodes into a flat Vec of operands.
118 /// 'A - B' is treated as 'A' and '(-B)'.
119 /// 'ak - ck - ax' -> '[ (ak), (-ck), (-ax) ]'
120 fn local_collect_operands(node: AstNode, operands: &mut Vec<AstNode>) {
121     match node {
122         // Case: A + B
123         AstNode::BinaryOperation {
124             operation: BinaryOperationKind::Plus,
125             left,
126             right,
127         } => {
128             // Collect operands from both sides
129             Self::local_collect_operands(*left, operands);
130             Self::local_collect_operands(*right, operands);
131         },
132         // Case: A - B
133         AstNode::BinaryOperation {
134             operation: BinaryOperationKind::Minus,
135             left,
136             right,
137         } => {
138             // Collect operands from the left side
139             Self::local_collect_operands(*left, operands);
140             // Collect operands from the right side, but wrap them in UnaryMinus
141             Self::local_collect_operands_with_minus(*right, operands);
142         },
143         // Base case: A standalone term (like 'ak' or '-ck')
144         _ => {
145             operands.push(node);
146         },
147     }

```

```

148     }
149
150     /// Helper for 'local_collect_operands' to correctly apply UnaryMinus.
151     /// This handles '-(A+B) -> -A + -B' and '-(-A) -> A'.
152     fn local_collect_operands_with_minus(node: AstNode, operands: &mut Vec<AstNode>) {
153         match node {
154             // Case: -(-A) => A
155             AstNode::UnaryOperation {
156                 operation: crate::compiler::ast::tree::UnaryOperationKind::Minus,
157                 expression,
158             } => {
159                 Self::local_collect_operands(*expression, operands);
160             },
161             // Case: -(A + B) => -A, -B
162             AstNode::BinaryOperation {
163                 operation: BinaryOperationKind::Plus,
164                 left,
165                 right,
166             } => {
167                 Self::local_collect_operands_with_minus(*left, operands);
168                 Self::local_collect_operands_with_minus(*right, operands);
169             },
170             // Case: -(A - B) => -A, B
171             AstNode::BinaryOperation {
172                 operation: BinaryOperationKind::Minus,
173                 left,
174                 right,
175             } => {
176                 Self::local_collect_operands_with_minus(*left, operands);
177                 Self::local_collect_operands(*right, operands); // Note: right side becomes positive
178             },
179             // Base case: -(term) => push(-term)
180             _ => {
181                 operands.push(AstNode::UnaryOperation {
182                     operation: crate::compiler::ast::tree::UnaryOperationKind::Minus,
183                     expression: Box::new(node),
184                 });
185             },
186         }
187     }
188
189     /// Builds a balanced tree from a list of operands.
190     /// (This is a standard helper function, same as in balancer.rs)
191     fn local_build_balanced_tree(
192         operands: Vec<AstNode>, op_kind: BinaryOperationKind,
193     ) -> Result<AstNode, AstError> {
194         if operands.is_empty() {
195             return Err(AstError::CannotBuildEmptyTree);
196         }
197         let mut queue: VecDeque<AstNode> = operands.into();

```

```

198     while queue.len() > 1 {
199         let level_size = queue.len();
200         for _ in 0..(level_size / 2) {
201             let left = queue.pop_front().ok_or(AstError::FailedPopFromQueue)?;
202             let right = queue.pop_front().ok_or(AstError::FailedPopFromQueue)?;
203             let new_node = AstNode::BinaryOperation {
204                 operation: op_kind.clone(),
205                 left: Box::new(left),
206                 right: Box::new(right),
207             };
208             queue.push_back(new_node);
209         }
210         if !level_size.is_multiple_of(2) {
211             let odd_one_out =
212                 queue.pop_front().ok_or(AstError::FailedPopFromQueue)?;
213             queue.push_back(odd_one_out);
214         }
215     }
216     queue.pop_front().ok_or(AstError::FailedPopFromQueue)
217 }
218
219     /// Helper function: finds the factors for a term.
220     /// 'A * B' -> '[A, B]'
221     /// '-(A * B)' -> '[A, B]' (sign is handled by get_remainder)
222     fn get_factors(node: &AstNode) -> Vec<AstNode> {
223         match node {
224             // Case: A * B
225             AstNode::BinaryOperation {
226                 operation: BinaryOperationKind::Multiply,
227                 left,
228                 right,
229             } => {
230                 vec![*left.clone(), *right.clone()]
231             },
232             // Case: -(A * B)
233             AstNode::UnaryOperation {
234                 operation: crate::compiler::ast::tree::UnaryOperationKind::Minus,
235                 expression,
236             } => {
237                 if let AstNode::BinaryOperation {
238                     operation: BinaryOperationKind::Multiply,
239                     left,
240                     right,
241                 } = expression.as_ref()
242                 {
243                     // Factors are A and B. The minus sign is part of the "remainder".
244                     vec![*left.clone(), *right.clone()]
245                 } else {
246                     vec![]
247                 }

```

```

248     },
249     _ => vec![], // Not a multiplication, no factors
250 }
251
252
253     /// Helper function: for 'node' (term) and 'factor', returns the remainder.
254     /// '(A * B, A)' -> 'B'
255     /// '(-(A * B), A)' -> '(-B)'
256     fn get_remainder(node: &AstNode, factor: &AstNode) -> Option<AstNode> {
257         match node {
258             // Case: A * B
259             AstNode::BinaryOperation {
260                 operation: BinaryOperationKind::Multiply,
261                 left,
262                 right,
263             } => {
264                 if left.as_ref() == factor {
265                     return Some(*right.clone()); // (A * B) / A = B
266                 }
267                 if right.as_ref() == factor {
268                     return Some(*left.clone()); // (A * B) / B = A
269                 }
270                 None
271             },
272             // Case: -(A * B)
273             AstNode::UnaryOperation {
274                 operation: op @ crate::compiler::ast::tree::UnaryOperationKind::Minus,
275                 expression,
276             } => {
277                 if let AstNode::BinaryOperation {
278                     operation: BinaryOperationKind::Multiply,
279                     left,
280                     right,
281                 } = expression.as_ref()
282                 {
283                     if left.as_ref() == factor {
284                         // -(A * B) / A = -B
285                         return Some(AstNode::UnaryOperation {
286                             operation: op.clone(),
287                             expression: right.clone(),
288                         });
289                     }
290                     if right.as_ref() == factor {
291                         // -(A * B) / B = -A
292                         return Some(AstNode::UnaryOperation {
293                             operation: op.clone(),
294                             expression: left.clone(),
295                         });
296                     }
297                 }
298             }
299         }
300     }

```

```

298         None
299     },
300     _ => None,
301   }
302 }
303 }
304
305

```

## distributive.rs

```

1  use crate::compiler::ast::tree::{AbstractSyntaxTree, AstNode, BinaryOperationKind};
2
3  // A "path" is a sequence of 0s and 1s (and 2s for Unary)
4  // 0 = left child, 1 = right child, 2 = unary expression
5  type NodePath = Vec<u8>;
6
7  impl AbstractSyntaxTree {
8    /// Returns a vector of AbstractSyntaxTree, each representing a single-step expansion.
9    pub fn get_all_single_step_expansions(&self) -> Vec<AbstractSyntaxTree> {
10      let mut expandable_nodes_paths: Vec<NodePath> = Vec::new();
11
12      // 1. Find all paths to nodes that can be expanded
13      Self::find_expandable_nodes_recursive(
14        &self.peek,
15        &mut vec![],
16        &mut expandable_nodes_paths,
17      );
18
19      let mut all_forms = Vec::new();
20
21      // 2. For each found path...
22      for path in expandable_nodes_paths {
23        // Create a fresh copy of the tree for this single expansion
24        let mut new_tree = self.clone();
25
26        // 3. Get a mutable reference to the node at that path
27        if let Some(target_node) =
28          Self::get_node_mut_by_path(&mut new_tree.peek, &path)
29        {
30          // 4. Replace that node with its expanded version
31          // 'perform_expansion' is guaranteed to expand *this* node
32          *target_node = Self::perform_expansion(target_node.clone());
33        }
34        all_forms.push(new_tree);
35      }
36
37      all_forms
38    }
39
40    /// Recursively finds nodes matching the pattern 'A * (B +/- C)' or '(A +/- B) * C'

```

```

41 fn find_expandable_nodes_recursive(
42     node: &AstNode, path: &mut NodePath, paths: &mut Vec<NodePath>,
43 ) {
44     if let AstNode::BinaryOperation {
45         operation: BinaryOperationKind::Multiply,
46         left,
47         right,
48     } = node
49     {
50         // Pattern 1: (A +/- B) * C
51         if let AstNode::BinaryOperation { operation: op, .. } = left.as_ref()
52             && (*op == BinaryOperationKind::Plus || *op == BinaryOperationKind::Minus)
53         {
54             // This 'Multiply' node can be expanded. Save its path.
55             paths.push(path.clone());
56         }
57         // Pattern 2: A * (B +/- C)
58         if let AstNode::BinaryOperation { operation: op, .. } = right.as_ref()
59             && (*op == BinaryOperationKind::Plus || *op == BinaryOperationKind::Minus)
60         {
61             // This 'Multiply' node can also be expanded. Save its path.
62             paths.push(path.clone());
63         }
64     }
65
66     // Recursive traversal to check children
67     match node {
68         AstNode::BinaryOperation { left, right, .. } => {
69             path.push(0); // 0 = left
70             Self::find_expandable_nodes_recursive(left, path, paths);
71             path.pop();
72
73             path.push(1); // 1 = right
74             Self::find_expandable_nodes_recursive(right, path, paths);
75             path.pop();
76         },
77         AstNode::UnaryOperation { expression, .. } => {
78             path.push(2); // 2 = expression
79             Self::find_expandable_nodes_recursive(expression, path, paths);
80             path.pop();
81         },
82         // Base cases (Number, Identifier): nowhere else to go
83         _ => {},
84     }
85 }
86
87     /// Helper function to get mutable reference to a node by path
88 pub fn get_node_mut_by_path<'a>(
89     node: &'a mut AstNode, path: &[u8],
90 ) -> Option<&'a mut AstNode> {

```

```

91     let mut current = node;
92     for &index in path {
93         match current {
94             AstNode::BinaryOperation { left, right, .. } => {
95                 current = if index == 0 { left } else { right };
96             },
97             AstNode::UnaryOperation { expression, .. } => {
98                 current = expression;
99             },
100            _ => return None, // Path is invalid
101        }
102    }
103    Some(current)
104 }
105
106 /// Performs ONE unfolding on a node that is *guaranteed* to be 'Multiply'
107 /// and have at least one child that is 'Plus' or 'Minus'.
108 fn perform_expansion(node: AstNode) -> AstNode {
109     if let AstNode::BinaryOperation {
110         operation: BinaryOperationKind::Multiply,
111         left,
112         right,
113     } = node
114     {
115         // Case 1: A * (B +/- C)
116         if let AstNode::BinaryOperation {
117             operation: op @ (BinaryOperationKind::Plus | BinaryOperationKind::Minus),
118             left: b, // B
119             right: c, // C
120         } = *right
121         {
122             // Create (A * B)
123             let new_left = AstNode::BinaryOperation {
124                 operation: BinaryOperationKind::Multiply,
125                 left: left.clone(), // A
126                 right: b,
127             };
128             // Create (A * C)
129             let new_right = AstNode::BinaryOperation {
130                 operation: BinaryOperationKind::Multiply,
131                 left, // A
132                 right: c,
133             };
134             // Return (A*B) +/- (A*C)
135             return AstNode::BinaryOperation {
136                 operation: op,
137                 left: Box::new(new_left),
138                 right: Box::new(new_right),
139             };
140     }

```

```

141
142     // Case 2: (A +/- B) * C
143     if let AstNode::BinaryOperation {
144         operation: op @ (BinaryOperationKind::Plus | BinaryOperationKind::Minus),
145         left: a, // A
146         right: b, // B
147     } = *left
148     {
149         // Create (A * C)
150         let new_left = AstNode::BinaryOperation {
151             operation: BinaryOperationKind::Multiply,
152             left: a,
153             right: right.clone(), // C
154         };
155         // Create (B * C)
156         let new_right = AstNode::BinaryOperation {
157             operation: BinaryOperationKind::Multiply,
158             left: b,
159             right, // C
160         };
161         // Return (A*C) +/- (B*C)
162         return AstNode::BinaryOperation {
163             operation: op,
164             left: Box::new(new_left),
165             right: Box::new(new_right),
166         };
167     }
168
169     // If patterns didn't match (e.g., this was called on a wrong node)
170     // return the original node.
171     return AstNode::BinaryOperation {
172         operation: BinaryOperationKind::Multiply,
173         left,
174         right,
175     };
176 }
177
178 // Return the node as is if it's not 'Multiply'
179 node
180 }
181 }
182

```

## equivalent\_forms.rs

```

1 use crate::compiler::ast::tree::AbstractSyntaxTree;
2 use crate::compiler::reports::Reporter;
3 use crate::utils::StringBuffer;
4 use std::collections::{HashSet, VecDeque};
5
6 impl AbstractSyntaxTree {

```

```

7  pub fn find_equivalent_forms(&self) -> Vec<AbstractSyntaxTree> {
8      let mut all_forms: Vec<AbstractSyntaxTree> = Vec::new();
9      let mut visited: HashSet<String> = HashSet::new();
10     let mut expansion_queue: VecDeque<AbstractSyntaxTree> = VecDeque::new();
11
12     // --- Stage 1: Expansion (Distributive Law, Nodes 0-7 from memo) ---
13     // This stage finds all forms *only* by expanding parentheses.
14     // It performs a Breadth-First Search (BFS) starting from the original expression.
15
16     let initial_key = self.to_canonical_string();
17     expansion_queue.push_back(self.clone());
18     visited.insert(initial_key);
19     all_forms.push(self.clone());
20
21     let mut fully_expanded_node: Option<AbstractSyntaxTree> = None;
22
23     while let Some(current_ast) = expansion_queue.pop_front() {
24         // Get all possible next forms by applying *one step* of expansion
25         let expansion_steps = current_ast.get_all_single_step_expansions();
26
27         // If a node has no possible expansions, it's a "leaf" in this stage.
28         // We assume the first one we find is the fully expanded form (Node 7).
29         if expansion_steps.is_empty() && fully_expanded_node.is_none() {
30             fully_expanded_node = Some(current_ast.clone());
31         }
32
33         for expanded_ast in expansion_steps {
34             let key = expanded_ast.to_canonical_string();
35             if !visited.contains(&key) {
36                 visited.insert(key.clone());
37                 all_forms.push(expanded_ast.clone());
38                 expansion_queue.push_back(expanded_ast);
39             }
40         }
41     }
42
43     // --- Stage 1.5: Flattening ---
44     // We take the "fully expanded" node (Node 7) and apply unary minus
45     // rules like '-(A-B) -> -A+B' to get the *truly* flat form.
46     // This is the form you're looking for, which has no parentheses.
47
48     let Some(node_to_flatten) = fully_expanded_node else {
49         log::warn!(
50             "No fully expanded form (Node 7) found. Skipping Stage 1.5 and 2."
51         );
52         return all_forms;
53     };
54
55     let node_to_flatten_copy = node_to_flatten.clone();
56     let start_node_for_factoring =

```

```

57     match Self::transform_recursive(node_to_flatten.peek)
58     .and_then(Self::fold_recursive)
59     {
60         Ok(flattened_node_peek) => {
61             let flattened_ast =
62                 AbstractSyntaxTree::from_node(flattened_node_peek);
63             let key = flattened_ast.to_canonical_string();
64             if !visited.contains(&key) {
65                 // Add this new, truly flat form if it's unique
66                 visited.insert(key);
67                 all_forms.push(flattened_ast.clone());
68             }
69             flattened_ast // This is the new starting point for factoring
70         },
71         Err(e) => {
72             log::error!("Failed to flatten Node 7: {:?}. Using un-flattened.", e);
73             node_to_flatten_copy // Fallback to the un-flattened node
74         },
75     };
76
77     // --- Stage 2: Factoring (Associative Law, Nodes 7-13) ---
78     // This stage now starts from the *truly flat* form.
79
80     let mut factoring_queue: VecDeque<AbstractSyntaxTree> = VecDeque::new();
81     factoring_queue.push_back(start_node_for_factoring);
82     // We don't need to re-add to 'visited' or 'all_forms',
83     // as it was handled in Stage 1.5
84
85     while let Some(current_ast) = factoring_queue.pop_front() {
86         let factoring_steps = current_ast.get_all_single_step_factorings();
87
88         for factored_ast in factoring_steps {
89             let key = factored_ast.to_canonical_string();
90             if !visited.contains(&key) {
91                 visited.insert(key.clone());
92                 all_forms.push(factored_ast.clone());
93                 factoring_queue.push_back(factored_ast);
94             }
95         }
96     }
97
98     all_forms
99 }
100 }
101
102 impl Reporter {
103     pub fn finding_equivalent_form(&self, forms: &[String]) -> String {
104         let mut buffer = StringBuffer::default();
105
106         buffer.add_line(format!("Found {} equivalent forms!\n", forms.len()));

```

```
107
108     for (index, form) in forms.iter().enumerate() {
109         buffer.add_line(format!("{} {}", index + 1, form));
110     }
111
112     buffer.get()
113 }
114 }
115
```