

Касянчук Дмитро Павлович

Аспірант

Національний технічний університет України «Київський політехнічний інститут імені Ігоря Сікорського», м. Київ, Україна

ORCID: 0009-0004-2824-8232

kasianchukdp@gmail.com

ГІБРИДНИЙ МЕТОД ПОТОКОВОЇ ВІДКЛАДЕНОЇ ПОБУДОВИ СИНТАКСИЧНОГО ДЕРЕВА

Анотація. У статті розглянуто проблему побудови синтаксичних дерев у системах статичного аналізу програмного коду, де класична повна матеріалізація дерева розбору може призводити до значних витрат оперативної пам'яті. Запропоновано гібридний метод потокової відкладеної побудови синтаксичного дерева, який поєднує просторову ефективність поточкових підходів із можливістю навігації по матеріалізованих фрагментах дерева. На відміну від традиційної моделі, у якій синтаксичне дерево повністю формується до початку подальшої обробки, запропонований підхід передбачає матеріалізацію вузлів у міру того, як алгоритм-споживач просувається синтаксичним деревом і потребує доступу до його фрагментів. Основою методу є подієво-орієнтована модель синтаксичного аналізу, у якій базовий аналізатор генерує впорядкований потік подій розбору, а спеціалізований адаптер перетворює ці події на частково матеріалізовані вузли та керуючі маркери. Для координації між генерацією подій і процесом матеріалізації використовується обмежений проміжний буфер, що забезпечує FIFO-семантику, контроль використання пам'яті та коректну взаємодію між постачальником і споживачем. Особливу увагу приділено формалізації відкладених обчислень, механізму форсування обіцянок, коректному завершенню потокового процесу та обробці аварійних станів. Показано, що фактична просторова складність запропонованої моделі визначається глибиною поточного стеку розбору, розміром проміжного буфера та множиною матеріалізованих вузлів, які залишаються досяжними через активні посилання алгоритму-споживача. У найгіршому випадку просторова складність наближається до складності класичного підходу з повною матеріалізацією дерева, тоді як у типових практичних сценаріях обходу, коли активні посилання на вже оброблені вузли втрачаються, виділена під них пам'ять може бути вивільнена збирачем сміття ще до завершення аналізу всієї структури. Це зменшує пікове споживання пам'яті та дозволяє повторно використовувати її в межах того самого процесу аналізу.

Ключові слова: синтаксичне дерево, відкладені обчислення, потокова обробка, подієво-орієнтована модель, синтаксичний аналіз, просторова складність

Kasianchuk Dmytro

Postgraduate student

National Technical University of Ukraine «Ihor Sikorskyi Kyiv Polytechnic Institute», Kyiv, Ukraine

ORCID: 0009-0004-2824-8232

kasianchukdp@gmail.com

A HYBRID METHOD FOR STREAMING DEFERRED CONSTRUCTION OF A SYNTAX TREE

Abstract. The article addresses the problem of constructing syntax trees in static program code analysis systems, where the classical full materialization of a parse tree may lead to significant memory overhead. A hybrid method for streaming deferred construction of a syntax tree is proposed. The method combines the spatial efficiency of streaming approaches with the ability to navigate materialized fragments of the tree. Unlike the traditional model, in which a syntax tree is fully constructed before further processing begins, the proposed approach provides for node materialization as the consumer algorithm traverses the syntax tree and requires access to its fragments. The method is based on an event-driven parsing model, in which the underlying parser generates an ordered stream of parsing events, while a specialized adapter transforms these events into partially materialized nodes and control markers. To coordinate event generation and the materialization process, a bounded intermediate buffer is used. It provides FIFO semantics, memory usage control, and correct interaction between the producer and the consumer. Special attention is paid to the formalization of deferred computations, the mechanism of forcing promises, correct termination of the streaming process, and handling of failure states. It is shown that the actual space complexity of the proposed model is determined by the depth of the current parsing stack, the size of the intermediate buffer, and the set of materialized nodes that remain reachable through active references held by the consumer algorithm. In the worst case, the space complexity approaches that of the classical approach with full tree materialization. However, in typical practical traversal scenarios, when active references to already processed nodes are lost, the memory allocated for them can be reclaimed by the garbage collector before the analysis of the entire structure is completed. This reduces peak memory consumption and allows that memory to be reused within the same analysis process.

Keywords: syntax tree, deferred computations, streaming processing, event-driven model, parsing, space complexity

1. Вступ

Сучасні системи статичного аналізу та компіляції програмного забезпечення працюють із дедалі більшими обсягами вихідного коду, що висуває підвищені вимоги до ефективності синтаксичного аналізу та побудови дерев розбору. У класичних підходах синтаксичний аналізатор виконує повну матеріалізацію дерева розбору перед початком подальших етапів обробки. Хоча така модель спрощує реалізацію аналізаторів, вона

супроводжується значними витратами пам'яті та затримками, особливо при роботі з великими кодовими базами. Внаслідок цього проблема ефективної потокової побудови синтаксичних дерев із контрольованим використанням пам'яті стає одним із ключових викликів сучасних систем статичного аналізу програмного коду.

2. Аналіз останніх досліджень і публікацій

Розвиток сучасних програмних систем супроводжується стрімким зростанням обсягів вихідного коду та масовим переходом до розробки на базі великих монорепозиторіїв. Це створює велике навантаження на інструментальні засоби статичного аналізу, пошуку вразливостей та автоматизованого рефакторингу. Як показує досвід провідних технологічних компаній, зокрема дослідження архітектури інфраструктури статичного аналізу компанії Facebook, класичні засоби аналізу коду з повною матеріалізацією синтаксичного дерева вимагають завантаження всієї програми в оперативну пам'ять, через що не здатні ефективно масштабуватися при зростанні обсягу кодової бази[1]. Спроба завантаження та побудови синтаксичних дерев або графів викликів для всього репозиторію одночасно, робить такі інструменти обчислювально невідповідними. У масштабах сучасних проєктів це перетворює аналіз на тривалі нічні пакетні завдання, які не встигають за швидкістю внесення змін розробниками. Обмеження пов'язані з повною матеріалізацією синтаксичного дерева зумовлюють необхідність розробки нових архітектурних рішень, зокрема переходу до композиційного аналізу[1], який хоч частково і вирішує проблему масштабування для задач пошуку вразливостей, він має суттєві обмеження у контексті засобів автоматизованого рефакторингу та складних трансформацій коду.

Композиційний підхід передбачає аналіз окремих синтаксичних структур (напр. методів чи класів) з подальшою генерацією для них компактних абстрактних математичних «зведень» (summaries). Щойно таке зведення буде сформоване, пам'ять виділена під синтаксичне дерево цієї структури вивільняється, і використовується для наступних частин програми[2]. Проте для точного переписування коду інструмент повинен зберігати повну інформацію про оригінальну програму, включаючи всі без винятку токени програми – навіть ті, що не несуть безпосереднього семантичного навантаження (наприклад, коментарі, пробільні символи, дужки та інші синтаксичні розділювачі), що робить використання абстрактних зведень неможливим. Крім того, на рівні обробки окремого модуля композиційні аналізатори продовжують використовувати парадигму жадібної (повної) матеріалізації. Це не вирішує проблему пікового споживання пам'яті: під час синтаксичного розбору масивних синтаксичних структур (наприклад, аномально великих класів чи методів) система все одно стикається з критичним вичерпанням ресурсів ще до етапу генерації зведень.

Таким чином, максимальна зручність ієрархічної навігації по повністю матеріалізованому синтаксичному дереву досягається ціною втрати масштабованості. Складність або неможливість аналізу надвеликих наборів даних змушує розробників вдаватися до штучної фрагментації вхідного коду на дрібніші незалежні сегменти. Своєю чергою, такий механічний поділ руйнує цілісність синтаксичного та семантичного аналізу: втрачається глобальний контекст програмного коду, а обробка перехресних посилань між різними фрагментами вимагає розробки додаткових, архітектурно складних механізмів для кешування та синхронізації проміжного стану.

Для подолання обмежень масштабованості, притаманних повній матеріалізації, розробники традиційно звертаються до альтернативної парадигми – подієво-орієнтованого (event-driven) або потокового синтаксичного аналізу, архітектурним еталоном якого є сімейство SAX-подібних інтерфейсів (Simple API for XML)[3],[4]. У цій моделі, яка, зокрема, реалізована в екосистемі ANTLR через патерн Listeners[5], процес синтаксичного розбору відбувається без матеріалізації всього синтаксичного дерева в оперативній пам'яті. Натомість, під час розпізнавання токенів та проходження синтаксичних правил аналізатор послідовно генерує події про початок та завершення розбору кожної синтаксичної конструкції, які перехоплюються програмою-споживачем за допомогою відповідних обробників (так званих callback-функцій). Головною перевагою такої архітектури є зниження просторової складності: споживання оперативної пам'яті стає константним $O(1)$ або обмежується виключно максимальною глибиною вкладеності синтаксичного дерева $O(h)$, адже аналізатор не зберігає вузли вже оброблених частин файлу, а підтримує лише поточний стек розбору (шлях від кореня дерева до поточного вузла), причому пам'ять, виділена під цей вузол, вивільняється одразу після завершення обробки відповідної синтаксичної конструкції. Такий підхід дозволяє успішно обробляти файли практично необмеженого розміру. Проте, така ефективність використання обчислювальних ресурсів досягається ціною ускладнення реалізації бізнес-логіки та виникнення «інверсії контролю»[6]. Оскільки синтаксичне дерево не зберігається в оперативній пам'яті, контроль над потоком виконання повністю переходить від розробника до синтаксичного аналізатора. Програма-споживач стає пасивним приймачем подій, який втрачає синтаксичний та семантичний контекст між викликами. Для того щоб відновити інформацію про ієрархію (наприклад, визначити батьківську область видимості під час обробки глибоко вкладених структур), розробник змушений власноруч проєктувати складні зовнішні машини станів, імплементувати стеки та вручну синхронізувати контекстні змінні. Єдина бізнес-логіка обробки виявляється фрагментованою та розподіленою по численних ізольованих обробниках подій. Це перетворює реалізацію навіть базових операцій, таких як агрегація пов'язаних даних чи багатокроковий аналіз, на перевантажене завдання, суттєво ускладнюючи підтримку коду та збільшує ризик виникнення логічних помилок.

Архітектурною альтернативою, що дозволяє подолати «інверсію контролю», є перехід до ітеративної моделі синтаксичного аналізу (pull parsing)[7],[8]. У цій парадигмі аналізатор інкапсулюється у вигляді потокового ітератора або курсора, а контроль над потоком виконання повністю повертається до програми-споживача. Розробник отримує можливість ініціювати розбір дерева «за запитом», шляхом викликів

методів ітератора, він віддає аналізатору команду на розпізнавання та вилучення чергового вузла синтаксичного дерева. При кожному такому виклику аналізатор просувається по вхідному потоку токенів рівно настільки, щоб сформувати одну завершену конструкцію, після чого повертає управління основній програмі, призупиняючи свою роботу до надходження наступного запиту від клієнта. Такий підхід дозволяє зберегти низьке споживання оперативної пам'яті – $O(1)$ або $O(h)$ – оскільки цілісне дерево все ще не формується, проте бізнес-логіка стає лінійною та прогнозованою[9]. Програміст отримує можливість використовувати стандартні алгоритмічні конструкції мови (цикли, умовні переходи) та звичайні локальні змінні для збереження контексту між ітераціями. Це повністю позбавляє розробника необхідності проектувати допоміжні структури даних для відстеження ієрархії (наприклад, явні автомати станів або стеки контексту). Замість цього поточний етап синтаксичного аналізу природним чином фіксується самим потоком виконання програми-споживача: її циклами, умовними переходами, стеком викликів та локальною областю видимості. Проте, незважаючи на вирішення проблеми інверсії контролю, ітеративна модель зберігає суттєві недоліки потокового аналізу – односпрямованість та відсутність цілісної структури. Оскільки синтаксичні вузли обробляються виключно послідовно, а історія пройдених вузлів не зберігається, розробник позбавлений можливості довільного доступу до раніше пройдених елементів або їхніх сусідніх гілок. Це робить ітеративну модель менш ефективною для завдань, що вимагають багатопрохідного семантичного аналізу, встановлення семантичних зв'язків між розрізненими вузлами або складних структурних трансформацій. У таких випадках розробник постає перед вибором: або багаторазово повторювати розбір вхідного потоку, що суттєво знижує продуктивність, або власноруч акумулювати пройдені вузли, конструюючи повне синтаксичне дерево в оперативній пам'яті, що повністю нівелює перевагу низької просторової складності.

Незважаючи на значний розвиток методів синтаксичного аналізу та потокової обробки даних, проблема поєднання ефективної просторової складності поточних підходів із можливістю повноцінної навігації по матеріалізованому синтаксичному дереву залишається недостатньо дослідженою. Традиційні синтаксичні аналізатори, як правило, виконують повну матеріалізацію дерева розбору, що спрощує подальшу обробку та навігацію, однак супроводжується значними витратами пам'яті. Натомість потокові підходи дозволяють зменшити споживання пам'яті за рахунок відкладеної або часткової обробки структури, проте ускладнюють довільний доступ до вузлів дерева та підтримку цілісної ієрархічної моделі.

3. Мета і задачі дослідження

Метою роботи є розробка гібридного методу потокової відкладеної побудови синтаксичного дерева, який поєднує ефективну просторову складність поточних підходів із можливістю навігації по повністю матеріалізованому дереву. Метод повинен забезпечувати інкрементальну матеріалізацію вузлів у міру їх споживання без необхідності попереднього повного розгортання всієї структури у пам'яті, зберігаючи при цьому коректність ієрархічної структури синтаксичного дерева.

4. Результати дослідження

4.1 Обґрунтування подієво-орієнтованого підходу та формальна модель синтаксичного дерева

Теоретично, ідеальною архітектурною моделлю для відкладеної (лінійної) побудови синтаксичних дерев є ітеративна модель, оскільки вона гарантує збереження ініціативи обчислень за споживачем і мінімізує споживання пам'яті. Однак існуючі високопродуктивні інструменти синтаксичного розбору промислового рівня, зокрема ANTLR, не мають ефективної нативної підтримки такої моделі. З огляду на це, найбільш придатною архітектурною основою для реалізації керованої відкладеної матеріалізації є подієво-орієнтована (event-driven) модель синтаксичного аналізу на базі LL-аналізаторів. Завдяки механізму випереджального читання токенів LL-аналізатор здатний детерміновано обирати продукцію грамматики ще до початку фактичного розбору її внутрішнього вмісту. Унаслідок цього побудова синтаксичного дерева відбувається послідовно згори вниз – від батьківських конструкцій до дочірніх елементів. Ця властивість принципово відрізняє LL-підходи від висхідних LR-алгоритмів, у яких батьківський вузол може бути сформований лише після повної обробки та фізичної матеріалізації всіх дочірніх елементів. Натомість LL-аналізатор ідентифікує тип батьківської конструкції ще до початку розбору вкладених структур, що створює можливість для відкладеної матеріалізації дочірніх вузлів. Додатково, класичні LL-аналізатори та їх сучасні модифікації (зокрема ALL(*)) базуються на моделі детермінованого рекурсивного спуску та не потребують повернення до попередніх станів розбору, що усуває необхідність зберігання альтернативних гілок розбору або підтримки громіздких структур мемоізації[10], і робить LL-архітектуру добре сумісною з потоковою моделлю обробки даних та відкладеною побудовою дерева. На базовому рівні подієвий аналізатор виступає низькорівневим рушієм, що продукує безперервний потік подій синтаксичного розбору. На вищому рівні функціонує механізм гібридної відкладеної матеріалізації, який інтерпретує цей потік та динамічно формує фрагменти синтаксичного дерева у міру їх фактичного споживання.

Формально синтаксичне дерево визначається як орієнтоване кореневе дерево

$T = \langle V, E, r \rangle$, де:

- $V = \{v_1, v_2, v_3, \dots, v_n\}$ – множина вузлів;
- $E \subseteq V \times V$ – множина ребер (ієрархічних зв'язків типу «батько-нащадок»);
- $r \in V$ – виділений кореневий вузол дерева.

Кожен синтаксичний вузол $v \in V$ визначається через кортеж властивостей:

$$v = \langle \tau, \lambda, \pi, C \rangle, \#(1)$$

де:

- τ – тип вузла (термінальний або нетермінальний символ граматики або спеціалізований маркер τ_{error} для забезпечення можливості помилкоустійкого аналізу);
- λ – лексичне значення вузла;
- $\pi = \langle i_{start}, i_{end} \rangle$ – просторові координати вузла (абсолютні зміщення першого та останнього символів відповідної синтаксичної конструкції у тексті вихідного коду).
- C – функція, що інкапсулює спосіб доступу до безпосередніх нащадків поточного вузла.

У класичній моделі послідовність безпосередніх нащадків вузла $v \in V$ визначається через множину ребер

E як:

$$children(v) = \langle (v, v_i) \in E \rangle \#(2)$$

Вона повертає колекцію вузлів, які вже існують у пам'яті. За такої організації просторова складність зберігання дерева є лінійною відносно кількості вузлів, що при аналізі великих кодових баз може призводити до значних витрат оперативної пам'яті та ставати обмежувальним фактором масштабованості системи аналізу коду.

Для усунення зазначеного обмеження у запропонованій моделі застосовується підхід відкладеної (lazy) матеріалізації синтаксичного дерева: повна структура дерева не будується наперед, а вузли генеруються динамічно в момент звернення до них. Компонента нащадків C формалізується як лінива структура даних, що інкапсулює обіцянку породження дочірніх вузлів. Її форсування матеріалізує черговий елемент послідовності, залишаючи решту нащадків у відкладеному стані. Подальше існування матеріалізованих вузлів визначається семантикою досяжності: вузол зберігається лише за наявності активного посилання з боку алгоритму-споживача; інакше він стає кандидатом на видалення збирачем сміття.

Початковий стан цієї структури задається обіцянкою θ_1 , форсування якої породжує перший дочірній вузол та обіцянку генерації решти елементів послідовності. Оскільки дочірні вузли утворюють впорядковану послідовність, процес їх розгортання описується рекурсивно визначеним ланцюгом обіцянок θ_i , де загальний рекурентний крок форсування обіцянки для i -го ($i \geq 1$) елемента визначається як:

$$force(\theta_i) = \langle v_i, \theta_{i+1} \rangle, \#(3)$$

де:

1. v_i – i -й дочірній вузол, частково матеріалізований на рівні власних статичних атрибутів (τ, λ, π), тоді як його компонента нащадків C залишається відкладеним обчисленням θ_{v_i} , що інкапсулює процес породження піддерева;
2. θ_{i+1} – обіцянка генерації решти дочірніх вузлів.

Процес генерації завершується після досягнення кінця послідовності дочірніх вузлів, що формально моделюється як форсування граничної обіцянки:

$$force(\theta_{n+1}) = \varepsilon, \#(4)$$

де n – загальна кількість безпосередніх нащадків поточного вузла, а ε позначає порожній результат, що сигналізує про завершення генерації на поточному рівні ієрархії.

Відповідно, глобальна множина ребер графа E у цій моделі не матеріалізується явно, а розглядається як імпліцитно задане відношення. Її елементи (конкретні топологічні зв'язки) не існують у пам'яті заздалегідь, а породжуються динамічно: виникають виключно в процесі форсування відповідних обіцянок під час розгортання дерева. Така організація забезпечує, що жоден вузол не матеріалізується до моменту фактичного звернення до нього.

4.2 Операційна семантика відкладеної матеріалізації синтаксичного дерева

Нехай W – функція розгортання, що задає операційну семантику відкладеної матеріалізації синтаксичного дерева. На відміну від суто декларативного опису, який фіксує лише статичну ієрархію обіцянок, W задає детермінований порядок форсування відкладених обчислень θ_i . Цей порядок не є довільним: він відповідає логіці спадного LL-аналізу та прямому обходу дерева (pre-order). Таким чином, функція W однозначно визначає послідовність поетапної матеріалізації вузлів, забезпечуючи коректну відбудову ієрархічної структури результуючого синтаксичного дерева.

Вона визначається індуктивно залежно від типу та структури вхідного операнда.

Для термінального стану (порожньої послідовності ε):

$$W(\varepsilon) = \varepsilon, \#(5)$$

рекурсія завершується, функція повертає порожній результат.

Для відкладеної обіцянки (θ):

$$W(\theta) = W(force(\theta)) = W(\langle v_{head}, \theta_{tail} \rangle), \#(6)$$

функція ініціює форсування обіцянки, що генерує нову сутність (пару або ϵ), і рекурсивно застосовується до результату.

Для лінійної пари послідовності ($\langle v_{head}, \theta_{tail} \rangle$):

$$W(\langle v_{head}, \theta_{tail} \rangle) = W(v_{head}) \oplus W(\theta_{tail}), \#(7)$$

функція декомпозує лінію пару, рекурсивно застосовуючи операцію розгортання до обох її компонентів. Отримані в результаті дві лінії послідовності (від розгортання першого нащадка та від розгортання решти нащадків) об'єднуються між собою за допомогою оператора лінійної конкатенації.

Для частково матеріалізованого вузла v :

$$W(v) = \langle v \rangle \oplus W(C(v)), \#(8)$$

вузол v стає першим елементом поточної лінійної послідовності. До нього лінійно приєднується результат рекурсивного застосування функції розгортання W до його нащадків, які ініціалізуються через виклик внутрішньої властивості $C(v)$.

Оператор $\oplus$ позначає лінійну конкатенацію послідовностей. Його ключовою властивістю в запропонованій моделі є асиметрична операційна семантика: він є строгим (strict) щодо лівого операнда і відкладеним (lazy) щодо правого. У контексті визначення функції розгортання W це означає, що обчислення $W(\theta_{tail})$ (матеріалізація наступних нащадків), відкладається до повного завершення розгортання піддерева $W(v_{head})$ (матеріалізації першого нащадка).

4.3 Гібридна архітектура потокової відкладеної побудови синтаксичного дерева

В основі запропонованого гібридного методу лежить поєднання подієво-орієнтованого синтаксичного розбору та ієрархічного представлення синтаксичного дерева. Розроблена гібридна архітектура усуває компроміс між подієво-орієнтованою (потоковою) обробкою та повною матеріалізацією синтаксичного дерева. У запропонованій моделі базовий синтаксичний аналізатор генерує послідовність подій розбору $e \in \Sigma_E$, що перехоплюються проміжним адаптером. Адаптер трансформує ці події в частково ініціалізовані вузли та керуючі маркери, які є необхідними для забезпечення процесу відкладеного розгортання синтаксичного дерева. Замість безпосередньої передачі алгоритму-споживачу, сформовані елементи розміщуються в обмеженому проміжному буфері B . Зовнішньо система надає стандартний інтерфейс синтаксичного дерева. Під час навігації деревом і звернення до ще нематеріалізованого фрагмента, ініціюється виконавець матеріалізації M , який форсує відповідні відкладені обчислення (обіцянки). Їх виконання, у свою чергу, призводить до вилучення з буфера B рівно тієї кількості елементів, яка необхідна для побудови запитаної гілки дерева. У разі переповнення буфера, генерація нових подій синтаксичним аналізатором тимчасово призупиняється до моменту звільнення місця виконавцем матеріалізації. Така модель дозволяє обмежити просторову складність синтаксичного аналізу, зберігаючи при цьому повноцінний інтерфейс навігації по синтаксичному дереву.

Графічне подання запропонованої архітектури та розподіл відповідальності між її ключовими компонентами наведено на рис. 1.

Варто окремо відзначити розмежування потоків даних та керування на наведеній схемі. Суцільні лінії відображають основний потік даних та операції, що змінюють стан системи: від генерації подій синтаксичного розбору, через їх трансформацію проміжним адаптером і розміщення у буфері, до інстанціювання нових вузлів дерева виконавцем M . Натомість пунктирні лінії відображають керуючі залежності та механізми координації між компонентами системи. У підсистемі постачальника вони демонструють поширення сигналу призупинення розбору у випадку переповнення буфера. У підсистемі споживача пунктирні зв'язки підкреслюють, що алгоритм-споживач здійснює виключно навігацію по частково матеріалізованій структурі, не модифікуючи її безпосередньо. Розгортання нових гілок дерева ініціюється виконавцем M лише під час звернення до ще нематеріалізованих фрагментів структури. Розглянемо компоненти зазначеної схеми.

Лексичний аналізатор – компонент обробки вхідного потоку символів, що реалізує перший рівень відкладених обчислень. Він не завантажує весь текст у пам'ять, а послідовно зчитує символи та формує токени виключно за запитом синтаксичного аналізатора.

Базовий синтаксичний аналізатор, який отримує токени від лексичного аналізатора, розпізнає синтаксичні конструкції та генерує події синтаксичного розбору $e \in \Sigma_E$. Аналізатор виступає виключно як джерело подій, що відображають проходження граматичних правил; він не виконує безпосередньої побудови синтаксичного дерева, не взаємодіє з проміжним буфером і є архітектурно ізольованим від механізму відкладеної матеріалізації.

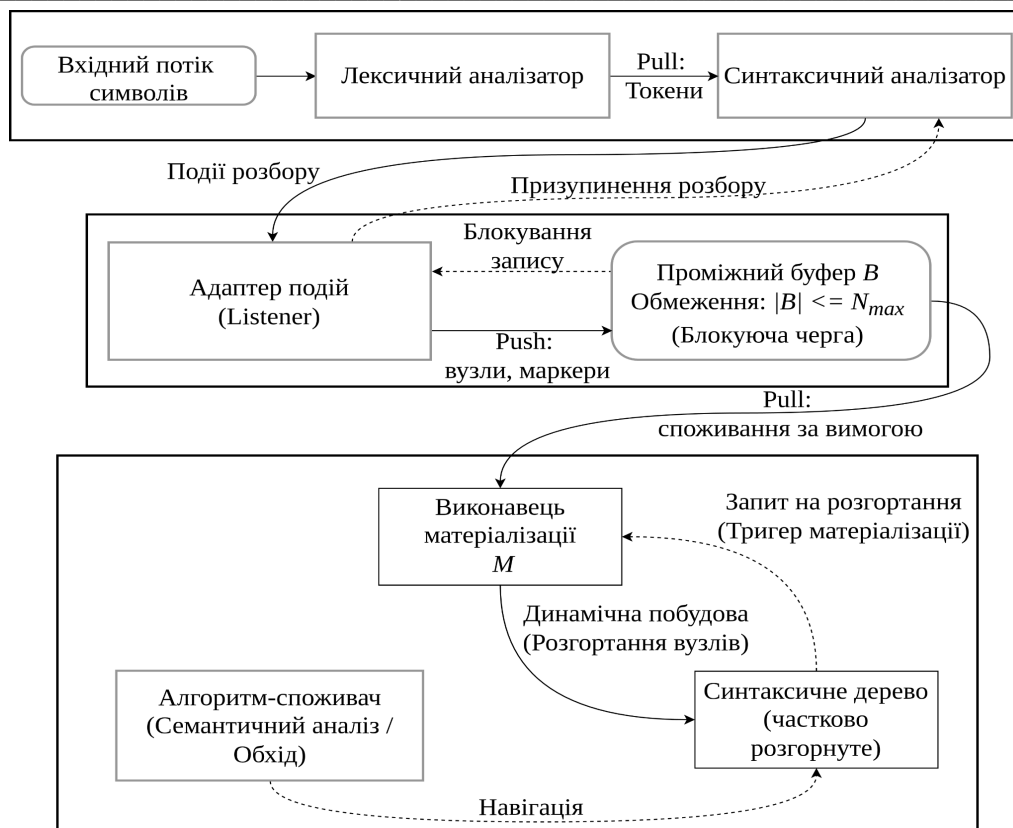


Рис. 1. Архітектура взаємодії компонентів гібридного синтаксичного аналізатора через проміжний буфер

Адаптер подій – проміжний компонент, що забезпечує інтеграцію між синтаксичним аналізатором та буфером B . Він виконує перехоплення подій синтаксичного розбору та конструює на їх основі об'єкти для проміжного буфера B : частково ініціалізовані вузли дерева із функцією відкладеної генерації нащадків та керуючі маркери. Крім цього, адаптер виконує функцію обмеження споживання пам'яті: у разі переповнення буфера B спроба запису блокує подальшу роботу адаптера. Оскільки адаптер розділяє з синтаксичним аналізатором спільний потік виконання, його призупинення транзитивно призупиняє весь процес синтаксичного розбору. Саме ця властивість запобігає передчасному повному зчитуванню вхідного потоку та гарантує, що синтаксичний розбір виконується виключно за вимогою, забезпечуючи відкладену матеріалізацію синтаксичного дерева.

Проміжний буфер (B) – потокобезпечна структура даних, реалізована у вигляді блокуючої черги (blocking queue) з фіксованою місткістю N_{max} . Буфер виконує роль асинхронної сполучної ланки між адаптером подій синтаксичного розбору і виконавцем матеріалізації M . Він тимчасово накопичує сформовані адаптером елементи до моменту їх безпосереднього запиту для розгортання дерева. Обмежений розмір буфера фізично контролює кількість елементів проміжного представлення, що можуть одночасно перебувати в оперативній пам'яті. Використання блокуючих операцій (призупинення запису у разі переповнення та очікування при читанні з порожньої черги) забезпечує узгодження швидкості роботи постачальника і споживача, дозволяючи підтримувати просторову складність проміжного зберігання на рівні $O(N_{max})$, незалежно від загального обсягу вхідного тексту.

Виконавець матеріалізації (M) – архітектурний компонент, що на фізичному рівні виконує відкладені обчислення (обіцянки), інкапсульовані адаптером у частково ініціалізовані вузли. Основне призначення виконавця полягає у реалізації порядку матеріалізації відповідно до операційної семантики функції W . Процес ініціюється алгоритмом-споживачем (pull-модель): під час звернення до списку нащадків певного вузла виконавець M послідовно ініціює відповідні відкладені обчислення. Це забезпечує інкрементальне інстанціювання дочірніх вузлів та їх перехід із відкладеного стану у повністю матеріалізовану ієрархічну структуру. Таким чином, M діє як виконавчий рівень системи, забезпечуючи те, що динаміка матеріалізації вузлів у пам'яті суворо відповідає порядку, визначеному семантикою W .

Синтаксичне дерево – цільова ієрархічна структура даних, яка виступає єдиним інтерфейсом взаємодії для алгоритму-споживача. У межах запропонованої моделі дерево перебуває у гібридному, частково матеріалізованому стані: воно складається з множини вже матеріалізованих вузлів, які кешуються в пам'яті для забезпечення ефективного повторного доступу, та нефорсованих обіцянок, які представляють ще нерозгорнуті піддерева. Для алгоритму-споживача система забезпечує повну прозорість: дерево надає стандартний інтерфейс навігації, приховуючи деталі відкладеного виконання. Звернення до ще нематеріалізованих фрагментів структури призводить до ініціювання виконавця M , який забезпечує подальшу розбудову дерева. Таке

представлення дозволяє уникнути повної матеріалізації дерева до початку його фактичного обходу: обсяг утримуваної в пам'яті структури обмежується лише тими вузлами, які алгоритм-споживач фактично розгорнув і продовжує утримувати за посиланнями, що робить можливим вивільнення пам'яті для вузлів, які перестають бути досяжними в процесі обходу.

Алгоритм-споживач – прикладний компонент (наприклад, семантичний аналізатор, транслятор або літер), який виступає основним ініціатором процесу матеріалізації та цільовим користувачем синтаксичного дерева. Діючи в рамках запропонованої моделі, споживач здійснює навігацію деревом через стандартний інтерфейс доступу, залишаючись повністю абстрагованим від механіки відкладених обчислень, проміжного буфера B та виконавця M . У процесі обходу гілки дерева матеріалізуються згідно з правилами операційної семантики W . Таким чином, алгоритм-споживач не лише використовує дані, а й опосередковано формує перебіг обчислень: його рішення щодо заглиблення в певні структури чи їх ігнорування, а також тривалість утримання посилань на відвідані вузли безпосередньо формують профіль споживання оперативної пам'яті та процесорного часу системою.

4.4 Алфавіт подій синтаксичного розбору та адаптер перетворення подій

Для забезпечення сумісності запропонованої моделі з широко використовуваними інструментами генерації синтаксичних аналізаторів, алфавіт подій Σ_E базується на класичній подієво-орієнтованій архітектурі (зокрема, на моделі Listener, що використовується в ANTLR). У цьому контексті алфавіт визначає стандартизований інтерфейс взаємодії між автоматом синтаксичного розбору та адаптером подій.

Основна ідея полягає в тому, що події розбору не формують ієрархічної структури та не містять явних структурних зв'язків між вузлами синтаксичного дерева. Вони є плоскими сигналами і відображають виключно зміну контексту розбору: заглиблення в нове правило, вихід із нього, розпізнавання токена або виявлення синтаксичної помилки. Такий підхід забезпечує компактне представлення кожної події та мінімізує додаткові витрати пам'яті. Згенеровані події не використовуються як довготривалі структури зберігання, а перехоплюються адаптером, який трансформує їх у частково ініціалізовані вузли та відкладені обчислення (обіцянки).

Алфавіт Σ_E задається як множина базових типів подій синтаксичного розбору:

$$\Sigma_E = \{enterRule, exitRule, visitTerminal, visitErrorNode, fatalError\} \#(9)$$

Кожна екземпляр події з цього алфавіту параметризується значенням p (payload), яке виступає як уніфікований контейнер контекстних даних, семантика якого залежить від типу події. Важливо відзначити, що незалежно від типу події, контейнер p не містить структурного елемента C (колекції нащадків), що гарантує плоску структуру потоку подій. Формування ієрархічної структури при цьому повністю виконується адаптером та механізмом відкладеної матеріалізації.

Кожна подія з наведеного алфавіту виконує чітко визначену семантичну функцію в межах загальної моделі.

$enterRule(p)$ – сигналізує про початок розбору нового нетермінального правила граматики. Ця подія фіксує входження синтаксичного аналізатора у вкладений контекст (наприклад, початок аналізу математичного виразу або блоку коду).

$exitRule(p)$ – сигналізує про успішне завершення розбору поточного нетермінального правила. Вона вказує на те, що синтаксичний аналізатор повністю розпізнав усі елементи цього правила і виходить із поточного контексту на рівень вище.

$visitTerminalNode(p)$ – позначає розпізнавання термінального символу (лексему або токена).

$visitErrorNode(p)$ – спеціальна подія, що генерується механізмом відновлення після синтаксичних помилок. Вона сигналізує про наявність невалідного або неочікуваного токена, дозволяючи зафіксувати факт помилки без переривання загального потоку розбору.

$fatalError(p)$ – спеціальна подія, що сигналізує про непередбачуване аварійне переривання процесу розбору. На відміну від $visitErrorNode$, вона не генерується внутрішньою логікою синтаксичного аналізатора, а виникає внаслідок критичних винятків у середовищі виконання (наприклад, системних збоїв або вичерпання ресурсів), де p містить об'єкт відповідної помилки. Ця подія формалізує стан аварійного завершення процесу розбору та забезпечує його коректну репрезентацію у вигляді події, навіть у випадках, коли сам синтаксичний аналізатор втрачає можливість подальшого виконання.

Таким чином, будь-який вхідний текст під час роботи синтаксичного аналізатора перетворюється на потік подій:

$$S = \langle s_1, s_2, s_3, \dots, s_n \rangle, \#(10)$$

$$s_i = e_i(p_i) \#(11)$$

де $e_i \in \Sigma_E$. Кожна подія обробляється в потоковому режимі та після перехоплення адаптером не зберігається у вигляді довготривалого стану.

Головне завдання адаптера полягає у трансформації плаского потоку подій синтаксичного розбору у проміжне структурне представлення, придатне для подальшої відкладеної матеріалізації синтаксичного дерева. Адаптер послідовно перехоплює кожний екземпляр події, базовий тип якої належить алфавіту Σ_E , що генерується синтаксичним аналізатором. Залежно від типу отриманої події та її параметрів p , адаптер або безпосередньо ініціалізує відповідний синтаксичний вузол, або інкапсулює контекст подальшої генерації нащадків у вигляді відкладеного обчислення. Сформовані вузли та керуючі маркери адаптер розміщує у проміжному буфері B , який виконує роль буферизованого каналу взаємодії між підсистемою розбору, виконавцем матеріалізації M та алгоритмом-споживачем.

Таким чином, логіка роботи адаптера формалізується у вигляді функції відображення:

$$f_A : \Omega_E \rightarrow (V \cup K), \#(12)$$

яка перетворює екземпляр події розбору у синтаксичний вузол або керуючий маркер, де:

1. Ω_E – множина всіх екземплярів подій, утворених параметризацією базових типів подій з алфавіту, які формалізуються наступним чином:

$$\Omega_E = \{s = e(p), e \in \Sigma_E\} \#(13)$$

2. V – множина синтаксичних вузлів, де синтаксичний вузол $n \in V$ формалізується як кортеж:

$$n = \langle \tau, \lambda, \pi, C \rangle \#(14)$$

3. $K = \{k_{stop}, k_{error}\}$ – множина керуючих маркерів, що не мають структурного представлення в дереві, але визначають контрольні точки процесу розбору, де:

- k_{stop} – структурний маркер, що сигналізує про завершення розбору поточного нетермінального правила (формування нащадків поточного вузла);
- k_{error} – спеціальний термінальний маркер, що інкапсулює об'єкт помилки p та сигналізує про настання фатального системного збою під час процесу синтаксичного аналізу.

Дія адаптера визначається типом вхідної події e :

$$f_A(e(p)) = \{n, \text{ if } e \in \Sigma_E \setminus \{exitRule, fatalError\} k_{stop}, \text{ if } e = exitRule k_{error}, \text{ if } e = fatalError \} \#(15)$$

Механізм відкладених обчислень при цьому застосовується вибірково: для нетермінальних вузлів адаптер формує відкладену генерацію нащадків, тоді як термінальні та помилкові вузли матеріалізуються одразу. Для детального розкриття цього процесу доцільно розглянути правила застосування функції f_A до кожного типу подій окремо.

Почнемо розбір з найпростіших сценаріїв – формування базових листових вузлів дерева. Коли адаптер перехоплює подію типу $visitTerminalNode(p)$, він інстанціює термінальний вузол n_{term} . Нехай параметр події, визначений наступним чином:

$$p = \langle \tau_p, \lambda_p, \pi_p \rangle, \#(16)$$

містить тип розпізнаного терміналу τ_p , його лексичне значення λ_p та просторові координати π_p . Тоді адаптер ініціалізує відповідні компоненти кортежу синтаксичного вузла. Оскільки термінальний вузол за визначенням не має нащадків, для збереження єдиного інтерфейсу поле C ініціалізується тривіальною функцією:

$$C_\epsilon : () \rightarrow \epsilon, \#(17)$$

яка завжди повертає порожню колекцію нащадків.

Отже, сформований термінальний вузол представляється кортежем:

$$n_{term} = \langle \tau_p, \lambda_p, \pi_p, C_\epsilon \rangle \#(18)$$

Відповідно, результат відображення адаптера для події синтаксичного аналізу, що відповідає розпізнаванню термінального символу граматики (токена), формалізується наступним чином:

$$f_A(visitTerminalNode(p)) = n_{term} \#(19)$$

Процес обробки події $visitErrorNode(p)$ виконується за правилами, аналогічними до інстанціювання термінального вузла. Оскільки, параметр події p інкапсулює токен, який спричинив синтаксичну помилку (наприклад, неочікуваний символ), адаптер аналогічно бере з нього лексичне значення λ_p та просторові координати π_p . Відмінність полягає у визначенні синтаксичного типу: попри наявність оригінального типу

токена в p , адаптер не використовує його та призначає вузлу фіксований тип τ_{error} . Таке перевизначення є суттєвим для семантики синтаксичного дерева, оскільки воно дозволяє алгоритму-споживачу однозначно відрізнити синтаксичну помилку від валідно розпізнаного терміналу без необхідності додаткового контекстного аналізу. Структурна компонента нащадків ініціалізується як C_ε .

Отже, сформований помилковий вузол представляється кортежем:

$$n_{error} = \langle \tau_{error}, \lambda_p, \pi_p, C_\varepsilon \rangle \# (20)$$

Відповідно, результат відображення адаптера для події синтаксичного аналізу, що генерується механізмом відновлення після синтаксичних помилок, формалізується наступним чином:

$$f_A(\text{visitErrorNode}(p)) = n_{error} \# (21)$$

Після розгляду механізмів формування базових листових вузлів перейдемо до механізму побудови структурних (нетермінальних) вузлів дерева, який ініціюється подією *enterRule*. При перехопленні цієї події адаптер отримує з її параметру ідентифікатор нетермінального правила. На відміну від термінальних вузлів, нетермінал не має власного лексичного значення, оскільки виконує виключно структурну функцію – визначає ієрархічну організацію та підпорядкованість своїх нащадків у межах синтаксичного дерева. Просторові координати такого вузла також не можуть бути остаточно визначені в момент його створення, адже відповідне піддерево ще не розгорнуте. У зв'язку з цим координати ініціалізуються у відкладеному стані та набувають конкретних значень лише в процесі матеріалізації нащадків.

Таким чином, адаптер формує частково матеріалізований нетермінальний вузол n_{rule} , який задається кортежем:

$$n_{rule} = \langle \tau_p, \varepsilon, \pi_{lazy}, \theta_1 \rangle, \# (22)$$

де:

1. τ_p – ідентифікатор нетерміналу (ім'я синтаксичного правила), отриманий з параметра події;
2. ε – маркер відсутності лексичного значення;
3. π_{lazy} – відкладені просторові координати;
4. θ_1 – початкова обіцянка генерації нащадків.

Відповідно, результат відображення адаптера для події синтаксичного аналізу, що сигналізує про початок розбору нового нетермінального правила граматики формалізується наступним чином:

$$f_A(\text{enterRule}(p)) = n_{rule} \# (23)$$

Формально процес форсування обіцянки θ_i визначається наступною системою співвідношень:

$$\text{force}(\theta_i) = \{\varepsilon, \text{if } b_i = k_{stop} \text{ halt}(p), \text{if } b_i = k_{error} \langle b_i, \theta_{i+1} \rangle, \text{if } b_i \in V, \# (24)$$

де b_i – елемент вилучений з буфера B .

Згідно з цією системою, форсування обіцянки призводить полягає в послідовному вилученні елементів із проміжного буфера B . Якщо отриманим елементом є синтаксичний вузол, результатом стає впорядкована пара, де перший компонент відповідає цьому вузлу, а другий – новій обіцянці θ_{i+1} , що інкапсулює продовження процесу розгортання послідовності нащадків. Цей процес завершується після вилучення з буфера B спеціального керуючого маркера k_{stop} , який сигналізує про завершення розбору поточного нетермінального правила. У цьому випадку форсування обіцянки повертає порожню послідовність ε . Згідно з операційною семантикою функції W , отримання ε інтерпретується виконавцем матеріалізації M як сигнал завершення матеріалізації поточного піддерева. Це ініціює завершення поточного рівня рекурсивного розгортання та повернення до батьківського контексту обробки, після чого продовжується подальше розгортання синтаксичного дерева. Важливо зазначити, що маркер k_{stop} не генерується безпосередньо синтаксичним аналізатором, а додається до буфера адаптером під час перехоплення події *exitRule*.

Відповідно, результат відображення адаптера для події завершення розбору нетермінального правила визначається наступним чином:

$$f_A(\text{exitRule}(p)) = k_{stop} \# (25)$$

$\text{halt}(p)$ – термінальний стан процесу матеріалізації, що моделює аварійне переривання обчислень з ретрансляцією діагностичної інформації або об'єкта помилки p . Перехід у цей стан при вилученні маркера k_{error} ініціює безумовну зупинку виконавця M та забезпечує властивість гарантованої завершуваності. Оскільки виконавець M функціонує як споживач, що зчитує події з буфера B за допомогою блокуючих операцій, йому

необхідний формальний механізм запобігання стану необмеженого очікування. У стані $halt(p)$ подальша обробка примусово припиняється незалежно від наявності інших елементів у черзі. Це усуває ризик переходу потоку виконавця M у стан нескінченного блокування у сценарії втрати або некоректного завершення роботи постачальника. Хоча за нормальних умов побудова синтаксичного дерева коректно фіналізується після обробки структурного маркера k_{stop} для кореневого вузла, маркер k_{error} виконує роль системного запобіжника. У випадку непередбачуваного припинення генерації вхідного потоку подій (наприклад, через фатальний збій синтаксичного аналізатора) процес матеріалізації переходить у $halt(p)$ без подальших блокуючих операцій і з коректним звільненням зайнятих ресурсів.

Розглядаючи процес побудови дерева в цілому, ініціатором його лінивого розгортання виступає перший нетермінальний вузол, сформований адаптером під час перехоплення початкової події синтаксичного розбору та розміщений у проміжному буфері B . Саме він стає коренем усього синтаксичного дерева, вилучення якого запускає роботу виконавця матеріалізації M . Подальша матеріалізація вузлів здійснюється відповідно до операційної семантики функції розгортання W .

4.5 Обчислення відкладених просторових координат нетермінальних вузлів

Безпосереднє обчислення відкладених просторових координат π_{lazy} для нетермінального вузла базується на покроковій агрегації координат його дочірніх вузлів у порядку їх матеріалізації.

Проте цей процес ускладнюється двома факторами:

1. не кожен нащадок має фізичне відображення у тексті. Типовим прикладом є обробка порожніх синтаксичних правил (ϵ -продукцій). Оскільки під час розпізнавання таких правил синтаксичний аналізатор не здійснює поглинання символів із вхідного потоку, сформовані для них вузли не мають фактичного відображення у вихідному коді. Відповідно, їхні просторові координати ініціалізуються порожніми значеннями $\pi = \langle \emptyset, \emptyset \rangle$.
2. через відкладений характер побудови дерева на момент вилучення дочірнього вузла з буфера B , його координати (у випадку нетермінала) перебувають у необчисленому стані π_{lazy} і не містять фінальних значень.

Для вирішення цієї проблеми пропонується механізм ітеративного оновлення просторових координат, що спирається на строгий послідовний порядок розгортання нащадків. Внутрішня логіка обіцянки θ_i розширюється таким чином, що після повернення пари $\langle n_i, \theta_{i+1} \rangle$, де $n_i = b_i$, наступна обіцянка θ_{i+1} отримує доступ до стану просторових координат щойно вилученого вузла n_i .

Для формального опису введемо послідовність станів координат батьківського вузла:

$$S_0, S_1, S_2, \dots, S_m, \#(26)$$

де $S_i = \langle start^{(i)}, end^{(i)} \rangle$, що відображає значення відкладених координат π_{lazy} після матеріалізації i -го нащадка.

Початковий стан перед матеріалізацією нащадків визначається як порожній:

$$S_0 = \langle \emptyset, \emptyset \rangle \#(27)$$

При форсуванні обіцянки θ_{i+1} виконується перехід до нового стану координат S_i на основі просторових координат попереднього матеріалізованого нащадка n_i , що задаються кортежем:

$$\pi_i = \langle start_i, end_i \rangle, \#(28)$$

Логіка покрокового оновлення відкладених координат π_{lazy} формалізується наступною системою рекурентних співвідношень:

$$start^{(i)} = \{ start_i, \text{ if } start^{(i-1)} = \emptyset \text{ AND } start_i \neq \emptyset, \text{ otherwise } \#(29) \}$$

$$end^{(i)} = \{ end_i, \text{ if } end_i \neq \emptyset, \text{ otherwise } \#(30) \}$$

Ця система гарантує, що ліва межа фіксується за першим нащадком, який має фізичне відображення у вихідному коді, тоді як права межа послідовно оновлюється при появі кожного нового непорожнього нащадка, ігноруючи при цьому ϵ -продукції.

Цей підхід є детермінованим завдяки операційній семантиці функції W . Згідно з її правилами, виконавець матеріалізації M форсує обіцянку θ_{i+1} лише після повного завершення розгортання піддерева вузла n_i . Отже, на момент переходу до наступного кроку обчислення, координати вузла n_i гарантовано є фіналізованими та можуть бути використані для оновлення координат батьківського вузла.

Процес покрокового формування координат завершується після вилучення з буфера B керуючого маркера k_{stop} . Перед поверненням порожньої послідовності ε координати батьківського вузла востаннє оновлюються значеннями останнього нащадка n_m . У цей момент відкладені координати π_{lazy} набувають фінального стану S_m , де m – загальна кількість нащадків:

$$\pi_{lazy} = S_m = \langle start^{(m)}, end^{(m)} \rangle \# (31)$$

4.6 Вимоги до проміжного буфера B

Розглянемо ключові вимоги, які висуваються до реалізації буфера B для забезпечення ефективної та стабільної роботи гібридного синтаксичного розбору.

Обмежена місткість та контроль використання пам'яті. Для забезпечення обмеженої просторової складності кількість елементів у буфері B у будь-який момент часу повинна задовольняти умову $|B| \leq N_{max}$, де N_{max} – максимально допустима місткість буфера. Це обмеження є критично важливим для збереження переваг відкладеної матеріалізації. За відсутності верхньої межі, у випадку коли адаптер генерує елементи швидше, ніж виконавець матеріалізації M зчитує їх з буфера під час розгортання дерева, кількість елементів у буфері може зростати пропорційно розміру вхідних даних. У такому випадку просторові переваги лінивого представлення дерева фактично втрачаються, що призводить до неконтрольованого зростання споживання пам'яті. Константа N_{max} виконує роль механізму балансування навантаження. При досягненні межі $|B| = N_{max}$ адаптер тимчасово призупиняє генерацію нових елементів до моменту, поки виконавець матеріалізації M не звільнить місце у буфері. Значення N_{max} може бути як статично заданою константою, так і динамічним параметром виконання. У складніших реалізаціях допускається адаптивна зміна розміру буфера на основі характеристик швидкості генерації та споживання елементів, що дозволяє балансувати між пропускнуою здатністю системи та рівнем споживання пам'яті.

Конкурентний доступ та семантика блокуючої черги. Оскільки адаптер, який додає згенеровані елементи, та виконавець матеріалізації M , який їх споживає, функціонують незалежно та можуть виконуватися у різних потоках, вони потребують безпечного конкурентного доступу до буфера B . Для узгодження швидкості генерації та споживання елементів, а також раціонального використання процесорного часу, буфер реалізується як обмежена блокуюча черга. Використання блокуючої семантики дозволяє уникнути неефективного активного очікування – ситуації, за якої потік безперервно перевіряє доступність ресурсу в циклі, витрачаючи процесорні ресурси без виконання корисної роботи. Замість цього потоки призупиняються до виконання відповідної умови у двох граничних станах:

1. при досягненні межі місткості буфера $|B| = N_{max}$, потік адаптера тимчасово блокується, очікуючи звільнення місця;
2. при виснаженні буфера $|B| = 0$, потік виконавця матеріалізації блокується в очікуванні на надходження нових елементів.

Важливо зазначити, що призупинення потоків у цих граничних станах є очікуваною поведінкою, визначеною семантикою архітектурного шаблону «постачальник–споживач», і не призводить до взаємного блокування. Гарантований вихід із режиму очікування забезпечується тим, що при оновленні вмісту буфера (вилученні елемента споживачем або додаванні нового елемента постачальником) виконання відповідного призупиненого потоку автоматично відновлюється.

Висока пропускну здатність та оптимізація паралельного доступу. Для забезпечення максимальної пропускну здатності системи у проміжних станах буфера, при $0 < |B| < N_{max}$ його внутрішня реалізація повинна мінімізувати накладні витрати на синхронізацію. Оскільки адаптер додає елементи виключно в кінець черги, а виконавець матеріалізації M вилучає їх з початку, структура даних має підтримувати оптимізовані механізми конкурентного доступу – зокрема, використання роздільних механізмів блокування для операцій запису та читання. Це дозволяє постачальнику та споживачу працювати з буфером паралельно, з мінімальною взаємною синхронізацією, зводячи до мінімуму конкуренцію потоків за доступ до спільного ресурсу під час виконання регулярних операцій. Додатково, для підвищення ефективності на сучасних багатоядерних архітектурах, структура буфера повинна враховувати особливості організації кеш-пам'яті процесора та мінімізувати ефект хибного розділення кешу (false sharing)[11]. Оскільки постачальник і споживач можуть працювати на різних процесорних ядрах, доцільно уникати ситуацій, за яких незалежно змінювані службові поля буфера розміщуються в межах однієї кеш-лінії, оскільки це призводить до додаткових витрат на узгодження стану кеш-пам'яті між процесорними ядрами. У практичних реалізаціях це досягається шляхом вирівнювання структур даних до меж кеш-ліній із використанням додаткового вирівнювального заповнення пам'яті (наприклад, cache line padding) або застосування спеціалізованих платформних примітивів синхронізації, які інкапсулюють механізми оптимізації конкурентного доступу на рівні середовища виконання.

Строге дотримання послідовності (FIFO-семантика). Для коректної побудови синтаксичного дерева критично важливо, щоб елементи вилучалися з буфера B виконавцем матеріалізації M суворо у тому ж порядку, в якому вони були згенеровані та розміщені адаптером. Оскільки буфер містить впорядковану послідовність частково матеріалізованих вузлів та керуючих маркерів, де відкладені обчислення батьківських вузлів

динамічно споживають наступні елементи з черги для генерації власних нащадків, виконавець матеріалізації M повинен обробляти цей потік строго лінійно. Будь-яке порушення цього порядку призводить до неправильного формування ієрархічних зв'язків між батьківськими та дочірніми вузлами. Таким чином, внутрішня архітектура буфера B повинна гарантувати строгий FIFO-порядок, унеможливаючи будь-яке переупорядкування елементів навіть в умовах конкурентного доступу з боку паралельних потоків.

Ізоляція стану та імутабельність елементів. Буфер B повинен містити виключно імутабельні елементи, повністю ізольовані від внутрішнього стану синтаксичного аналізатора. Будь-які мутабельні структури даних, сформовані аналізатором (зокрема, об'єкти контексту розбору), мають бути попередньо трансформовані адаптером у імутабельні об'єкти до моменту їх публікації у буфері. Ця вимога виключає виникнення станів гонитви (race conditions) при конкурентному доступі. Додатково, відсутність прямих посилань на внутрішні структури синтаксичного аналізатора запобігає утриманню об'єктів у пам'яті та дозволяє механізму керування пам'яттю своєчасно звільняти ресурси, які не використовуються.

4.7 Просторова складність запропонованої моделі

Просторова складність запропонованої моделі визначається фактичним обсягом матеріалізованої частини синтаксичного дерева. У кожен момент часу пам'ять витрачається на три основні компоненти: глибину поточного стеку розбору h , елементи проміжного буфера B , кількість яких обмежена константою N_{max} , та множину матеріалізованих вузлів $V_{target} \subseteq V$, на які алгоритм-споживач зберігає активні посилання. Тоді фактичні просторові витрати можна оцінити як:

$$O\left(h + |V_{target}| + N_{max}\right) \quad (32)$$

У найгіршому випадку, коли алгоритм-споживач утримує посилання на кореневий вузол та ініціює повну матеріалізацію всієї структури, просторова складність зводиться до класичної лінійної оцінки $O(|V|)$.

Таким чином, запропонована модель забезпечує адаптивну просторову поведінку: її просторова складність не перевищує складності класичного підходу з повною матеріалізацією синтаксичного дерева, але за сприятливих сценаріїв обходу може бути істотно меншою. Якщо в процесі аналізу окремі вузли стають недосяжними, тобто на них більше не існує активних посилань з боку алгоритму-споживача, пам'ять, виділена під ці об'єкти, може бути вивільнена збирачем сміття ще до завершення аналізу всієї структури. Це зменшує пікове споживання пам'яті та дозволяє повторно використовувати обчислювальні ресурси в межах того самого процесу аналізу.

5. Висновки та перспективи подальших досліджень

У роботі запропоновано гібридний метод потокової відкладеної побудови синтаксичного дерева, що поєднує просторову ефективність поточних підходів із можливістю навігації по матеріалізованих фрагментах дерева. На відміну від класичних моделей, у яких синтаксичне дерево повністю будується в пам'яті до початку подальшої обробки, запропонований підхід передбачає матеріалізацію вузлів у міру того, як алгоритм-споживач просувається синтаксичним деревом і потребує доступу до його фрагментів. Ключовим елементом моделі є поділ процесу обробки на два рівні: базовий синтаксичний аналізатор, що генерує впорядкований потік подій розбору, та механізм відкладеної матеріалізації, який інтерпретує цей потік і формує ієрархічну структуру дерева за потребою. Для забезпечення коректності такого процесу в роботі формалізовано роль адаптера, буфера B , керуючих маркерів, обчислювальних об'єктів та функції операційної семантики W , яка визначає порядок виконання відкладених обчислень. Запропонована модель дозволяє контролювати просторові витрати системи. Фактичне споживання пам'яті визначається глибиною поточного стеку розбору, розміром проміжного буфера та множиною матеріалізованих вузлів, що залишаються досяжними через активні посилання алгоритму-споживача. У найгіршому випадку, коли дерево повністю матеріалізується і залишається досяжним через кореневий вузол, просторова складність зводиться до класичної оцінки $O(|V|)$. Водночас у типових практичних сценаріях обходу, коли активні посилання на вже оброблені вузли втрачаються, виділена під них пам'ять може бути вивільнена збирачем сміття ще до завершення аналізу всієї структури. Це зменшує пікове споживання пам'яті та дозволяє повторно використовувати її в межах того самого процесу аналізу.

Одним із напрямів подальшого розвитку запропонованого методу є його практична програмна реалізація та експериментальна апробація. Оскільки запропонована модель спирається на відкладені обчислення, перспективним інструментарієм для реалізації виконавчого рівня системи є функціональні мови програмування. Зокрема, подальші дослідження доцільно зосередити на розробці експериментального інструменту статичного аналізу коду з використанням мови програмування Clojure [12]. Її архітектурні особливості, зокрема підтримка лінійних обчислень та розвинені примітиви асинхронної взаємодії, відповідають ключовим вимогам запропонованої теоретичної моделі. У ролі генератора подій синтаксичного аналізу доцільною є інтеграція інструментарію ANTLR [5] як популярного представника спадного LL -аналізу. Наступним етапом має стати проведення серії експериментальних вимірювань для підтвердження теоретичних оцінок просторової ефективності розробленого методу.

Декларація про штучний інтелект

Автор декларує, що інструменти штучного інтелекту (Chat GPT) застосовувалися виключно для допоміжного редагування тексту (перевірка стилістики, граматики та формулювань). Штучний інтелект не використовувався для генерування наукового змісту, результатів дослідження, інтерпретації даних або формулювання висновків. Усі наукові положення статті є результатом самостійної роботи автора.

Конфлікт інтересів

Автор заявляє про відсутність конфлікту інтересів та підтверджує, що під час підготовки цієї роботи не існувало жодних комерційних, фінансових чи інших взаємовідносин, які могли б бути розцінені як такі, що здатні вплинути на результати дослідження або їх інтерпретацію. Робота виконана відповідно до принципів академічної доброчесності, етичних норм проведення наукових досліджень та вимог редакційної політики щодо запобігання конфлікту інтересів.

Список використаної літератури

1. Distefano, D., Fähndrich, M., Logozzo, F., & O'Hearn, P. W. (2019). *Scaling static analyses at Facebook*. *Communications of the ACM*, 62(8), 62–70. <https://doi.org/10.1145/3338112>
2. Calcagno, C., Distefano, D., O'Hearn, P. W., & Yang, H. (2011). *Compositional Shape Analysis by Means of Bi-Abduction*. *Journal of the ACM*, 58(6), 1–66. <https://doi.org/10.1145/2049697.2049700>
3. Brownell, D. (2002). *SAX2*. O'Reilly Media.
4. Wu, D., Chau, K. T., Wang, J., & Pan, C. (2019). *A comparative study on performance of XML parser APIs (DOM and SAX) in parsing efficiency*. *U the 3rd International Conference*. ACM Press. <https://doi.org/10.1145/3309074.3309124>
5. Parr, T. (2012). *The definitive ANTLR 4 reference*. Pragmatic Bookshelf.
6. Bainomugisha, E., Carreton, A. L., Cutsem, T. v., Mostinckx, S., & Meuter, W. d. (2013). *A survey on reactive programming*. *ACM Computing Surveys*, 45(4), 1–34. <https://doi.org/10.1145/2501654.2501666>
7. Kay, M. *You Pull, I'll Push: on the Polarity of Pipelines*. In *Balisage: The markup conference 2009*. Mulberry Technologies, Inc. <https://doi.org/10.4242/balisagevol3.kay01>
8. Pulgatti, L., & Didonet Del Fabro, M. (2018). *JSON-based Interoperability Applying the Pull-parser Programming Model*. In *20th International Conference on Enterprise Information Systems*. SCITEPRESS - Science and Technology Publications. <https://doi.org/10.5220/0006646400950102>
9. Müller, J. (2023). *A fast, compliant JSON pull parser for writing robust applications [Доповідь на конференції]*. CppCon 2023. YouTube. https://www.youtube.com/watch?v=_GrHKyUYyRc
10. Parr, T., Harwell, S., & Fisher, K. (2014). *Adaptive LL(*) parsing*. *ACM SIGPLAN Notices*, 49(10), 579–598. <https://doi.org/10.1145/2714064.2660202>
11. Khan, T. A., Zhao, Y., Pokam, G., Mozafari, B., & Kasikci, B. (2019). *Huron: hybrid false sharing detection and repair*. In *PLDI '19: 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM. <https://doi.org/10.1145/3314221.3314644>
12. Hickey, R. (2020). *A history of Clojure*. *Proceedings of the ACM on Programming Languages*, 4(HOPL), 1–46. <https://doi.org/10.1145/3386321>

References

1. Distefano, D., Fähndrich, M., Logozzo, F., & O'Hearn, P. W. (2019). *Scaling static analyses at Facebook*. *Communications of the ACM*, 62(8), 62–70. <https://doi.org/10.1145/3338112>
2. Calcagno, C., Distefano, D., O'Hearn, P. W., & Yang, H. (2011). *Compositional Shape Analysis by Means of Bi-Abduction*. *Journal of the ACM*, 58(6), 1–66. <https://doi.org/10.1145/2049697.2049700>
3. Brownell, D. (2002). *SAX2*. O'Reilly Media.
4. Wu, D., Chau, K. T., Wang, J., & Pan, C. (2019). *A comparative study on performance of XML parser APIs (DOM and SAX) in parsing efficiency*. *U the 3rd International Conference*. ACM Press. <https://doi.org/10.1145/3309074.3309124>
5. Parr, T. (2012). *The definitive ANTLR 4 reference*. Pragmatic Bookshelf.
6. Bainomugisha, E., Carreton, A. L., Cutsem, T. v., Mostinckx, S., & Meuter, W. d. (2013). *A survey on reactive programming*. *ACM Computing Surveys*, 45(4), 1–34. <https://doi.org/10.1145/2501654.2501666>
7. Kay, M. *You Pull, I'll Push: on the Polarity of Pipelines*. In *Balisage: The markup conference 2009*. Mulberry Technologies, Inc. <https://doi.org/10.4242/balisagevol3.kay01>
8. Pulgatti, L., & Didonet Del Fabro, M. (2018). *JSON-based Interoperability Applying the Pull-parser Programming Model*. In *20th International Conference on Enterprise Information Systems*. SCITEPRESS - Science and Technology Publications. <https://doi.org/10.5220/0006646400950102>
9. Müller, J. (2023). *A fast, compliant JSON pull parser for writing robust applications [Conference presentation]*. CppCon 2023. YouTube. https://www.youtube.com/watch?v=_GrHKyUYyRc
10. Parr, T., Harwell, S., & Fisher, K. (2014). *Adaptive LL(*) parsing*. *ACM SIGPLAN Notices*, 49(10), 579–598. <https://doi.org/10.1145/2714064.2660202>

11. Khan, T. A., Zhao, Y., Pokam, G., Mozafari, B., & Kasikci, B. (2019). *Huron: hybrid false sharing detection and repair*. In *PLDI '19: 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM. <https://doi.org/10.1145/3314221.3314644>
12. Hickey, R. (2020). *A history of Clojure*. *Proceedings of the ACM on Programming Languages*, 4(HOPL), 1–46. <https://doi.org/10.1145/3386321>.

Надійшла до редакції: 27.03.2026

Прийнята до друку: 15.09.2026

Опубліковано: 30.09.2026