Автоматизація технологічних і бізнес-процесів Volume 18, Issue 1/2026
ISSN 2312-3125 (print), ISSN 2312-931X (online) http://www.atbp.ontu.edu.ua/

УДК 004.02+004.43

МЕТОД АНАЛІЗУ ТА ТРАНСФОРМАЦІЇ СИНТАКСИЧНИХ ДЕРЕВ

METHOD OF ANALYSIS AND TRANSFORMATION OF SYNTAX TREES

Касянчук Д.П.
Kasianchuk D.

НТУ України «Київський політехнічний інститут імені Ігоря Сікорського» м. Київ, Україна
National Technical University of Ukraine "Igor Sikorsky Kyiv Polytechnic Institute", Kyiv, Ukraine
ORCID: https://orcid.org/0009-0004-2824-8232
Email: kasianchukdp@gmail.com

DOI: 10.15673/atbp.v18i1.3430

Анотація. *Аналіз і трансформація абстрактних синтаксичних дерев є ключовими задачами, що виникають при розробці компіляторів, статичних аналізаторів та засобів автоматизованого рефакторингу. Розробники, які обирають мову Clojure для цих задач, стикаються з проблемою відсутності цілісної вбудованої системи для комплексного аналізу коду. Використання зовнішніх інструментів створюють значні проблеми при їх інтеграції, адже процеси аналізу та трансформації виконуються за межами основної програмної системи, що ускладнює спільне використання даних, тоді як наявні вбудовані рішення змушують працювати на низькому рівні абстракції або створюють «декларативно-імперативний розрив», де пошук є декларативним, а генерація нових дерев залишається імперативною. У роботі для вирішення цієї проблеми запропоновано метод, який реалізований шляхом створення вбудованої предметно-орієнтованої мови DALT. Суть методу полягає у структурній трансформації програмного коду, що базується на механізмі квазіцитування конкретного синтаксису. Цей підхід дозволяє оперувати не об'єктами синтаксичних дерев, а текстовими шаблонами, що візуально відповідають коду цільової мови. Ключовою особливістю методу є уніфікація процесів аналізу та синтезу: запропонований механізм шаблонів використовується для представлення синтаксичних дерев як в операціях зіставлення зі зразком, так і для генерації нових структур. У статті також описано розроблені декларативні засоби для зіставлення, рекурсивного переписування та синтезу дерев. Особливу увагу приділено гнучкому механізму керування стратегіями обходу, що дозволяє комбінувати правила з низхідною та висхідною семантикою в межах єдиного процесу трансформації. Ефективність методу підтверджено експериментально на задачі побудови графів потоку виконання для програм написаних мовою Java. Порівняльний аналіз продемонстрував, що застосування мови DALT дозволило суттєво скоротити обсяг коду порівняно з імперативною реалізацією.*

Abstract. *Analysis and transformation of abstract syntax trees are key tasks in the development of compilers, static analyzers, and automated refactoring tools. Developers choosing Clojure for these tasks face the lack of a comprehensive built-in system for complex code analysis. The use of external tools poses significant integration challenges, as analysis and transformation processes are executed outside the main software system, complicating data sharing. Meanwhile, existing built-in solutions force developers to work at a low abstraction level or create a «declarative-imperative gap» where search is declarative, but the generation of new trees remains imperative. To address this issue, this paper proposes a method implemented as an embedded domain-specific language named DALT. The core of the method lies in the structural transformation of program code based on the mechanism of concrete syntax quasiquoting. This approach allows manipulating text templates that visually correspond to the target language code, rather than operating directly on syntax tree objects. A key feature of the method is the unification of analysis and synthesis processes: the proposed template mechanism is used to represent syntax trees both in pattern matching operations and for generating new structures. The paper also describes the developed declarative tools for matching, recursive rewriting, and tree synthesis.*

Particular attention is given to a flexible traversal strategy control mechanism, which allows combining rules with top-down and bottom-up semantics within a single transformation process. The effectiveness of the method is experimentally validated on the task of constructing control flow graphs for Java programs. Comparative analysis demonstrated that the application of DALT significantly reduced the code volume compared to an imperative implementation.
Ключові слова: Clojure; предметно-орієнтована мова; аналіз дерева; трансформація дерева; зіставлення зі зразком
Keywords: Clojure; domain-specific language; tree analysis; tree transformation; pattern matching

Вступ

Аналіз і трансформація абстрактних синтаксичних дерев (AST) є фундаментальними етапами розробки компіляторів, систем статичного аналізу та засобів автоматизованого рефакторингу. Традиційно для реалізації цих процесів застосовуються спеціалізовані зовнішні інструменти. Однак їх використання породжує проблему складної інтеграції: процеси обробки коду виконуються відокремлено від основної системи, що ускладнює обмін даними.

Альтернативний підхід полягає у використанні засобів інструментальної мови загального призначення. Проте пряма реалізація алгоритмів обходу та модифікації дерев за допомогою імперативних конструкцій призводить до надлишкової складності коду. Логіка аналізу «розмивається» численними перевірками типів вузлів та деталями навігації, що суттєво ускладнює супровід та масштабування системи.

Попри те, що функціональні мови – зокрема Clojure [1] – надають потужні засоби для роботи з деревоподібними структурами, ці можливості залишаються розрізненими та не об'єднаними в цілісну систему засобів та, головне, не пристосованими для роботи з синтаксичними деревами. Крім того, інтеграція цих засобів із сучасними генераторами синтаксичних аналізаторів, такими як ANTLR [2], відсутня або обмежена, що ускладнює побудову повноцінних систем для аналізу та трансформації коду. Зазначені проблеми ускладнюють використання мови Clojure для вирішення задач пов'язаних з аналізом і трансформацією синтаксичних дерев.

У зв'язку з цим, актуальним є завдання розробки вбудованої предметно-орієнтованої мови (embedded DSL, eDSL [3]). Використовуючи можливості метапрограмування та макросистеми Clojure, така мова дозволяє абстрагуватися від технічних деталей реалізації AST. Це забезпечує уніфікацію процесів зіставлення зі зразком та трансформації дерев, гарантуючи при цьому повну інтероперабельність з екосистемою інструментальної мови.

Аналіз літературних даних

Задачі аналізу та трансформації абстрактних синтаксичних дерев є критично важливими для широкого спектру напрямків програмної інженерії: від розробки компіляторів до створення систем статичного аналізу та автоматизованого рефакторингу. За час розвитку галузі було розроблено значну кількість інструментальних засобів. Для їх систематизації доцільно застосувати класифікацію за критерієм рівня інтеграції з базовим середовищем розробки.

Існуючі рішення можна поділити на дві основні категорії:

1. Автономні мови (external/standalone DSLs), що являють собою окремі, повноцінні екосистеми, які мають власний синтаксис, семантику та середовище виконання. Вони надають високоспеціалізовані, формально обґрунтовані механізми, однак їх використання вимагає організації міжпроцесної взаємодії, серіалізації/десеріалізації даних та синхронізації етапів збірки. Це ускладнює інтеграцію інструменту в єдиний програмний комплекс.
2. Вбудовані рішення (embedded/internal DSLs), які реалізуються у вигляді бібліотек або предметно-орієнтованих мов безпосередньо всередині існуючої мови програмування (інструментальної мови). Такий підхід має на меті усунути проблему інтеграції, надаючи прямий доступ до всієї екосистеми інструментальної мови та розширюють її синтаксис і семантику для вирішення предметної задачі.

У цьому розділі ми розглянемо представників обох категорій, оцінюючи їхні переваги та недоліки в контексті уніфікації всього процесу обробки коду – від синтаксичного аналізу до трансформації.

Першу категорію рішень, яку ми розглянемо, складають автономні мови. Яскравим представником цієї категорії є TXL[4] – одна з найбільш ранніх та впливових мов, розроблена спеціально для завдань трансформації вихідного коду.

Парадигма TXL базується на переписуванні дерев за заданою граматикою. Робочий процес чітко розділений: спочатку розробник описує граматику мови, що аналізується, а потім визначає набір правил трансформації. Ці правила мають декларативну форму [match pattern] ... [replace by pattern] (знайти зразок ... замінити на зразок). Процесор TXL застосовує цей набір правил рекурсивно до всього синтаксичного дерева, доки не буде досягнуто «фіксованої точки», тобто стану, коли жодне з правил більше не може бути застосованим.

Переваги TXL:

1. Ключовою перевагою TXL є його націленість на конкретний синтаксис. Зразки – це не просто абстрактні структури даних, вони є спеціалізованими виразами записаними мовою, що схожа на мову оригінального коду. Це робить написання правил інтуїтивним, оскільки розробник описує структуру, що максимально наближена до того фрагменту коду, який він хоче знайти.

2. TXL довів свою ефективність у великомасштабних задачах промислового рефакторингу та міграції кодових баз, де потрібно застосувати тисячі однакових змін.

Недоліки TXL:

1. Як автономний інструмент, TXL працює в ізольованому середовищі. Інтеграція з іншими системами (зокрема, Clojure) вимагає організації міжпроцесної взаємодії та обміну даними через файлову систему. Це вносить суттєві затримки і ускладнює побудову інтерактивних інструментів (наприклад, плагінів для IDE).
2. Надає дуже обмежені засоби для точного керування стратегією трансформації, тому описати складний, багатопрохідний аналіз, де порядок застосування правил має вирішальне значення, в TXL складно.

Логічним розвитком ідей, закладених у TXL, стала мова Stratego, яка також належить до категорії автономних мов, але вона вирішує ключову проблему обмеженого контролю, наявну в TXL, шляхом запровадження парадигми стратегічного переписування. Актуальність цього підходу підтверджується його використанням у сучасній платформі Spoofax для вирішення промислових задач мовної інженерії, як це продемонстровано в [5] роботі.

Цей підхід базується на чіткому розділенні логіки трансформації на два рівні:

1. Правила переписування для опису локальних, атомарних змін у вузлах дерева. Вони декларативні і не містять інформації про те, де і в якому порядку їх застосовувати.
2. Стратегії – спеціалізовані комбінатори (функції вищого порядку), які керують потоком виконання. Вони визначають алгоритм обходу дерева (наприклад, top-down, bottom-up) та порядок застосування правил.

Переваги Stratego:

1. На відміну від TXL, де застосування правил часто відбувається автоматично до досягнення фіксованої точки, стратегії дозволяють розробнику явно програмувати алгоритм обходу. Це критично важливо для реалізації складних оптимізацій у компіляторах, де порядок трансформацій впливає на коректність результату.
2. Модель Stratego чітко відокремлює атомарну логіку правил від логіки керування.

Недоліки Stratego:

1. Stratego вимагає вивчення специфічної DSL зі складною семантикою комбінаторів, що створює високий поріг входження для інженерів.
2. як і TXL, Stratego є окремою екосистемою, і хоча вона надає API для взаємодії (наприклад, з Java), вона залишається автономним інструментом, що ускладнює налагодження та динамічну взаємодію з даними основної програми в реальному часі.

Окрему нішу серед автономних інструментів займає Rascal MPL (Meta-Programming Language), який виходить за рамки простої трансформації (як TXL) чи стратегічного переписування (як Stratego) і позиціонується як комплексна платформа для метапрограмування та аналізу коду. Ефективність інструментарію доведена на практиці, зокрема у складних задачах міграції великої кількості застарілого коду [6]. Rascal спроєктований так, щоб замістити собою інструментальне середовище, а не інтегруватися в нього. Замість того щоб бути бібліотекою, яку можна підключити до існуючого проєкту, він вимагає перенесення логіки обробки у власну, закриту екосистему. Це створює класичну проблему інтеграції: неможливість прямого виклику функцій бізнес-логіки основної програми.

Другу категорію рішень, яку ми розглянемо, складають вбудовані в програмну систему рішення.

Одним з відомих представників цієї категорії є TOM[7]. Це мова, розроблена для додавання гнучких конструкцій зіставлення зі зразком до мов, де вони відсутні (наприклад, Java, C, OCaml). Архітектурно TOM реалізує механізм зіставлення зі зразком не через засоби інструментальної мови, наприклад макроси, а через зовнішній препроцесор. Розробник інтегрує у вихідний код спеціалізовані блоки %match(term) { pattern => { <Java code> }, які описують правила зіставлення. На етапі збірки препроцесор транслює ці конструкції в код інструментальної мови (наприклад, каскади умовних переходів Java), який згодом компілюється стандартним компілятором. Оскільки конструкції TOM містять код інструментальної мови, то забезпечується прямий доступ до всіх бібліотек, змінних та стану програми. Це вирішує проблему ізольованості, притаманну автономним системам.

З недоліків TOM можна зазначити наступне:

1. Використання зовнішнього препроцесора ускладнює збірку. Більш критичною є проблема втрати відповідності коду, адже повідомлення про помилки компілятора та налагодження відбуваються в контексті згенерованого, часто нечитабельного коду, а не оригінального.
2. TOM є декларативним лише в аспекті пошуку. Логіка трансформації (дії при збігу) залишається чисто імперативною. Це створює концептуальний розрив: розробник декларує «що знайти», але змушений імперативно описувати «як замінити» (наприклад, через мутацію об’єктів), що знижує надійність та читабельність коду.

Еталонним прикладом «чистого» вбудовування, що не залежить від зовнішніх генераторів коду чи препроцесорів, є бібліотека Kiama [8] для мови Scala. Архітектурно Kiama адаптує парадигму стратегічного переписування (запозичену зі Stratego) до об’єктно-функціональної моделі Scala. Kiama повністю використовує засоби мови Scala для своєї реалізації, зокрема незмінні (англ. immutable) класи для опису вузлів дерева і їх

ієрархії, а також вбудоване зіставлення зі зразком. Вона реалізує ключові ідеї Stratego, надаючи функції вищого порядку (стратегії), для декларативного керування процесом трансформації.

З переваг Kiama можна зазначити наступне:

1) відсутність етапу кодогенерації спрощує процес збірки та відлагодження;
2) сильна типізація Scala дозволяє виявляти помилки ще на етапі компіляції.

Недоліком можна назвати те, що розробник змушений працювати на більш детальному рівні безпосередньо з класами окремих вузлів AST, а не з шаблонами описаними наближено до мови оригінального коду. Це створює семантичний розрив: правила трансформації стають громіздкими і візуально далекими від коду, що аналізується.

Мета і завдання дослідження

Метою дослідження є створення методу обробки синтаксичних дерев для підвищення ефективності розробки засобів аналізу та модифікації програмного коду на основі механізму квазіцитування (quasi-quotation), та проєктування відповідної вбудованої предметно-орієнтованої мови (eDSL) у середовищі Clojure.

Для досягнення мети необхідно вирішити наступні завдання:

1. Розробити метод структурної трансформації програмного коду, який базується на механізмі квазіцитування, що дозволяє уніфікувати представлення зразків пошуку та правил генерації нових дерев.
2. Сформулювати вимоги до архітектури та функціональності нової мови на основі проведеного аналізу недоліків існуючих інструментів (зокрема, проблем інтеграції та розриву між пошуком і дією).
3. Спроєктувати синтаксис та семантику конструкцій предметно-орієнтованої мови, що забезпечують високорівневу абстракцію над структурою синтаксичного дерева та приховують складність його обходу.
4. Продемонструвати ефективність запропонованого підходу на прикладі вирішення типової задачі статичного аналізу коду.

Методи і матеріали досліджень

Запропонована предметно-орієнтована мова DALT (Declarative AST Language & Transformations) є вдосконаленою версією інструментальних засобів, описаних у роботі [9]. Якщо попереднє рішення фокусувалося виключно на процесі синтаксичного розбору, то розроблена eDSL суттєво розширює цю базу, надаючи декларативний інструментарій для аналізу та модифікації отриманих дерев.

В основу DALT покладено метод структурної трансформації програмного коду на базі квазіцитування (quasi-quotation) конкретного синтаксису (concrete syntax). На відміну від класичних підходів, де робота з AST вимагає імперативного створення об’єктів та ручного обходу ієрархії класів, цей метод дозволяє оперувати фрагментами коду цільової мови.

Ключовим елементом методу є синтаксичний шаблон. Тобто, як в операціях синтезу, так і аналізу, операнди-дерева описуються синтаксичними шаблонами. Формально шаблон визначається як фрагмент вихідного коду цільової мови, в якому окремі синтаксичні конструкції заміщено на типізовані змінні шаблону, які описуються наступною граматикою:

$$\begin{aligned} templateVarName &::= \$[digit \mid letter \mid \_ \mid -]+ \\ digit &::= [0-9] \\ letter &::= [a-zA-Z] \end{aligned}$$

наприклад, “int $a-1 = $expr_2 + 1”.

Ці змінні позначають динамічні частини структури, роль яких змінюється залежно від контексту операції, і завдяки яким реалізується семантика квазіцитування.

Змінні шаблону можуть мати один з двох типів:

1) тип синтаксичного елемента граматики (нетермінала); змінна такого типу містить дерево, що репрезентує цілісну синтаксичну конструкцію структура якої відповідає певному правилу граматики (наприклад, expression);
2) тип лексичного елемента граматики; змінна такого типу містить значення токена, що репрезентує атомарну лексичну одиницю, яка відповідає певному термінальному символу граматики (наприклад, IDENTIFIER);

Тип визначається спираючись на конвенцію іменування ANTLR: імена синтаксичних елементів (нетерміналів) записуються маленькими літерами, а лексичних (токенів) – великими. Відповідно, тип ідентифікується за регістром першої літери його імені.

Суть методу полягає у контекстно-залежній трансляції синтаксичних шаблонів у часткове синтаксичне дерево (partial AST) – структуру, що відтворює ієрархію AST, але змінні шаблону перетворюються у спеціальні мета-вузли («пропуски») замість конкретних піддерев.

Семантика цих вузлів визначається контекстом операції:

1) у контексті синтезу (конструювання) змінні шаблону перетворюються у вузли підстановки. Під час виконання згенерованого коду вони замінюються на синтаксичні дерева, що містяться у відповідних змінних поточного лексичного контексту. Ці змінні можуть бути визначені різними шляхами: як результат попередньої операції зіставлення (збережені піддерева) або як результат довільних обчислень.

2) у контексті аналізу (зіставлення) ті ж самі змінні шаблону діють як вузли прив'язки. Вони перетворюються у логіку перевірки типів та екстракції даних: у разі успішної операції зіставлення, піддерева зберігаються у відповідних змінних, які вводяться у лексичний контекст і стають доступними для подальшої обробки.

Реалізація методу виконана у вигляді бібліотеки мови Clojure. Такий вибір зумовлений властивістю гомоіконності мови (представлення коду як структури даних) та її потужною системою макросів. Це дозволяє реалізувати механізми квазіцитування та маніпуляції з абстрактними синтаксичними деревами найбільш природним чином, оперуючи ними як стандартними списками, а не громіздкими об'єктами.

Для структурної організації компонентів введено чітку конвенцію найменування, що розділяє систему на два шари:

1. функціональне ядро (префікс «fsa-» – Functional Specification for AST) – базовий набір функцій для взаємодії з генератором синтаксичних аналізаторів ANTLR. Забезпечує синтаксичний аналіз вихідного коду та побудову початкового AST.
2. декларативний шар (префікс «msa-» – Macro Specification for AST) – система макросів, що реалізує механізм квазіцитування та стратегічного переписування. Важливою архітектурною особливістю реалізації є те, що трансляція синтаксичних шаблонів у часткові дерева виконується на етапі макророзширення. Це дозволяє уникнути додаткових витрат на розбір шаблонів під час виконання програми (runtime), оскільки структура AST формується ще на етапі компіляції.

Для забезпечення консистентності, функції, описані в [9] (antlr, register-lang, parse-source), були уніфіковані відповідно до прийнятої конвенції та перейменовані у fsa-generate-parser, fsa-register-lang та fsa-parse-source.

Ініціалізація та налаштування середовища аналізу у функціональному ядрі здійснюється через виклик базових функцій:

1. функція fsa-generate-parser забезпечує генерацію лексичного та синтаксичного аналізаторів на основі файлів граматики.
2. реєстрація згенерованих модулів здійснюється через функцію fsa-register-lang, яка створює асоціативний зв'язок між логічним ідентифікатором мови та згенерованими класами лексичного і синтаксичного аналізаторів.
3. функція fsa-use-lang дозволяє фіксувати поточну граматику для подальшого використання.

Безпосередній розбір програмного тексту у структуру даних здійснюється функцією fsa-parse-source. Вона ініціює процес синтаксичного розбору на основі попередньо зареєстрованої граматики. Її результатом є сформоване абстрактне синтаксичне дерево у внутрішньому представленні бібліотеки, яке є інваріантним до конкретної мови програмування та придатне для подальших маніпуляцій засобами DALT.

Для керування контекстом генерації в рамках декларативного шару (msa-) розроблено уніфікований формат оголошення змінних шаблону. Введення змінних у область видимості трансформації здійснюється через вектор визначень metaVarDefs. Ця структура являє собою послідовність, де описують ініціалізацію змінних.

Формально структура вектора визначень має наступний вигляд:

$$metaVarDefs \ ::= \ [metaVarDef*]$$
$$metaVarDef \ ::= \ metaVarType \ \ templateVarName \ \ metaVarExpr$$

де:

1. metaVarType – тип змінної шаблону. Явна специфікація типу є архітектурною вимогою, оскільки вона дозволяє коректно сформувати мета-вузли у частковому дереві на етапі трансляції шаблону, гарантуючи структурну цілісність результуючого AST.
2. templateVarName – ім'я змінної, яке буде використовуватись для посилань у шаблонах;
3. metaVarExpr – вираз мови Clojure або конструкція DALT. Критичною вимогою є те, що результат обчислення цього виразу повинен відповідати заявленому metaVarType та бути коректним синтаксичним деревом у внутрішньому представленні бібліотеки.

Для формалізації контексту аналізу використовується вектор декларацій bindVarsDecls. Він визначає набір змінних шаблону, призначених для збереження відповідних вузлів дерева в процесі зіставлення зі зразком.

З метою забезпечення компактності коду та усунення надмірності, формат декларації підтримує групове оголошення, дозволяючи асоціювати один тип з довільною кількістю змінних.

Формально структура вектора описується наступним чином:

$$bindVarsDecls \ ::= \ [bindVarDecl*]$$
$$bindVarDecl \ ::= \ bindVarType \ \ templateVarName+$$

де:

1. bindVarType – очікуваний тип вузла, якому має відповідати вузол з вхідного дерева. Тільки вузли вказаного типу можуть бути зв'язані зі змінними, що слідують за цим визначенням;
2. templateVarName+ – послідовність імен змінних шаблону. Всі змінні, вказані після типу (до наступного визначення типу або кінця вектора), успадковують обмеження bindVarType. Такий підхід дозволяє суттєво

скоротити опис однотипних елементів. Наприклад, декларація ”expression $left $right” типізує змінні $left та $right як вирази (expression), усуваючи необхідність повторення типу для кожної змінної окремо.

Також в DALT реалізовано механізм неявного виведення типів. Якщо змінна використовується у шаблоні, але не оголошена явно у списку bindVarsDecls, її тип вважається тотожним її імені (без урахування префікса $). Наприклад, змінна $block трактується як вузол типу :block, а змінна $expression – як вузол типу :expression. Такий підхід дозволяє уникнути дублювання інформації в коді, коли ім'я змінної збігається з назвою синтаксичної конструкції.

Для реалізації фази синтезу (побудови нових дерев) у мові DALT розроблено два взаємопов'язані макроси, що забезпечують лексичне зв’язування та генерацію структури.

Макрос msa-let реалізує механізм лексичного зв’язування змінних шаблону із синтаксичними деревами.

Сигнатура макроса:

$$(msa-let\ \ metaVarDefs\ \ \&body)$$

Макрос створює локальну область видимості, межі якої визначаються параметром body, що відповідає списку виразів мов Clojure і/або DALT. У цій області стають доступними змінні, оголошені у векторі metaVarDefs. Такий підхід забезпечує композиційність побудови синтаксичних дерев, дозволяючи формувати та налаштовувати окремі фрагменти коду (піддерева) перед їх фінальною інтеграцією в основну структуру.

Функцію конструктора абстрактного синтаксичного дерева виконує макрос msa-parse-pattern, що фактично реалізує семантику оператора квазіцитування.

Сигнатура макроса:

$$(msa-parse-pattern\ \ ruleName\ \ templateString)$$

Цей компонент відповідає за практичну реалізацію етапу трансляції шаблону згідно з методологією. Параметр templateString є текстовою репрезентацією синтаксичного шаблону. Під час макророзширення він трансформується у часткове синтаксичне дерево, причому синтаксичний розбір ініціюється зі стартового правила ruleName Сформована структура містить вузли підстановки, які в процесі виконання згенерованого коду заміщуються реальними значеннями змінних шаблону (піддеревами) із лексичного контексту.

На рис. 1 наведено приклад побудови дерева, яке відповідає операції присвоювання, що ілюструє взаємодію цих конструкцій. У цьому сценарії змінні $id та $expr, оголошені через msa-let позначають відповідні дерева, які інтегруються у шаблон "$id = $expr;". У результаті формується цілісна синтаксична конструкція, готова до подальшого використання.

```
(fsa-use-lang :java)

(msa-let [identifier $id (fsa-parse-source :identifier "a")
          expression $expr (fsa-parse-source :expression "b")]
 (msa-parse-pattern :statement "$id = $expr;"))
```

Рис. 1. Конструювання синтаксичного дерева операції присвоювання
Fig. 1. Constructing the syntax tree for an assignment operation

Для реалізації механізму структурного аналізу та декомпозиції AST розроблено макрос msa-match, що реалізує семантику зіставлення зі зразком.

Сигнатура макроса:

$$(msa-match\ \ treeExpr\ \ bindVarsDecls\ \ \&\ matchClauses)$$

де:

1. treeExpr – вираз, що обчислюється в об’єкт синтаксичного дерева (цільове AST);
2. bindVarsDecls – вектор декларацій змінних шаблону;
3. логіка зіставлення визначається послідовністю правил matchClauses, формальна структура яких описується граматикою:

$$\begin{aligned} matchClauses &::= matchClause^{*}\ defaultClause^{?} \\ matchClause &::= patternDescription\ action \\ patternDescription &::= (\ ruleName\ templateString\ ) \\ defaultClause &::= :else\ action \end{aligned}$$

Параметр templateString є текстовою репрезентацією синтаксичного шаблону, формалізованого у методології. Під час макророзширення він трансформується у часткове синтаксичне дерево, причому синтаксичний розбір ініціюється зі стартового правила ruleName (наприклад, :statement або :expression).

ISSN 2312-3125 (print), ISSN 2312-931X (online) http://www.atbp.ontu.edu.ua/

Вузли сформованого дерева класифікуються на дві функціональні категорії:
1. статичні вузли, які відповідають фіксованим частинам коду в шаблоні і слугують зразками, з якими порівнюються відповідні частини вхідного дерево;
2. вузли зв'язування, які асоційовані зі змінними, оголошеними у параметрі bindVarsDecls макроса. При обробці такого вузла відповідне піддерево зберігається та зв’язується зі змінною лише за умови рівності їх типів.

При успішному структурному зіставленні вхідного дерева зі зразком виконується вираз action. У тілі цього виразу стають доступними змінні, які містять відповідні фрагменти проаналізованого дерева. Гілка :else (якщо присутня) виконується лише тоді, коли жоден із зразків не збігся.

На рис. 2 наведено приклад практичного застосування макроса msa-match. У векторі декларацій змінні $id та $expr типізовано як IDENTIFIER (токен) та expression (нетермінал) відповідно.

Алгоритм роботи фрагменту наступний:
1. У разі структурної відповідності вхідного дерева зразку операції присвоювання, відбувається зв’язування змінних $id та $expr із відповідними піддеревами.
2. У тілі обробника виконується синтез результуючого AST за допомогою макроса msa-parse-pattern. При цьому формується нова конструкція операції присвоювання, де ліва частина ($id) залишається незмінною, а права частина ($expr) модифікується шляхом додавання одиниці.
3. Якщо структура вхідного дерева не відповідає зразку, виконується гілка за замовчуванням, що повертає літерал “Not matched”.

Таким чином, для вхідного синтаксичного дерева, яке є представленням виразу “a=b;”, результатом виконання цього фрагменту коду, буде синтаксичне дерево, яке є представленням виразу “a=b+1;”.

```
(fsa-use-lang :java)

(msa-match tree [IDENTIFIER $id
                 expression $expr]
(:statement "$id = $expr;")
(msa-parse-pattern :statement "$id = $expr + 1;")
:else "Not matched")
```

Рис. 2. Поєднання зіставлення зі зразком та синтезу нового синтаксичного дерева
Fig. 2. Combining pattern matching and construction of a new syntax tree

Для виконання задачі структурної трансформації був створений макрос msa-rewrite. Його призначення полягає у рекурсивному переписуванні вхідного синтаксичного дерева на нове синтаксичне дерево, яке буде сформоване шляхом зіставлення кожного піддерева вхідного дерева з одним із заданих зразків з подальшою їх заміною на нові піддерева.

Сигнатура макроса:

$$(msa-rewrite\ \ treeExpr\ \ bindVarsDecls\ \ \&\ \ rewriteClauses)$$

Макрос приймає наступні параметри:
1. treeExpr – вираз мови Clojure або DALT, який обчислюється в об’єкт синтаксичного дерева, що підлягає трансформації.
2. bindVarsDecls – вектор декларацій змінних шаблону.
3. rewriteClauses – сукупність правил трансформації. Кожне правило визначає зразок пошуку, стратегію обходу та логіку генерації нового піддерева.

Структура правил трансформації описується наступною граматикою:

$$\begin{aligned} rewriteClauses &::= rewriteClause* \\ rewriteClause &::= matchPattern\ \ newTreeExpr \\ matchPattern &::= (strategy\ patternDescription) \mid patternDescription \\ patternDescription &::= (ruleName\ \ templateString) \\ strategy &::= :pre \mid :post \end{aligned}$$

де:
1) newTreeExpr – вираз генерації (Clojure/DALT код). Він виконується лише у разі успішного зіставлення піддерева з відповідним зразком. Результатом виконання має бути нове синтаксичне дерево, яке заміщує знайдене. У межах цього виразу доступні змінні, які містять відповідні частини оригінального піддерева.
2) patternDescription – формат ідентичний параметру patternDescription макроса msa-match (див. вище).
3) strategy – ключове слово, що визначає фазу рекурсивного обходу, на якій відбувається порівняння зі зразком. Воно може набувати одного з двох значень:

• :pre (стратегія низхідного обходу / top-down) – перевірка відповідності зразку відбувається перед обробкою нащадків поточного вузла (на етапі рекурсивного спуску). Особливість полягає у тому, що у разі успішного зіставлення та заміни вузла, алгоритм не продовжує заглиблення у старі нащадки цього вузла. Обхід продовжується вже для сусідніх гілок. Це дозволяє замінювати цілі піддерева, уникаючи зайвих обчислень, та запобігає зацикленню при генерації нових структур, що містять шуканий зразок.

• :post (стратегія висхідного обходу / bottom-up) – перевірка відповідності зразку відбувається після того, як були оброблені всі нащадки поточного вузла (на етапі рекурсивного підйому). Особливість полягає у тому, що ця стратегія гарантує, що на момент перевірки поточного вузла всі його дочірні вузли вже пройшли етап трансформації. Це критично важливо для задач згортання виразів або обчислення значень знизу-вгору.

Якщо стратегія не вказана явно (використано форму patternDescription без обгортки), за замовчуванням застосовується стратегія :post.

В межах одного виклику msa-rewrite допускається оголошення гетерогенного набору правил у межах єдиного списку rewriteClauses. Це означає, що розробник може вільно чергувати правила з різними стратегіями. Правила трансформації з однаковою стратегією перевіряються послідовно у порядку їх оголошення на відповідному етапі рекурсивного обходу. Застосовується перше правило, зразок якого задовольняє структуру поточного вузла. Така архітектура дозволяє в межах одного виклику макроса реалізовувати складні алгоритми, що поєднують логіку декомпозиції (зверху-вниз) та згортання/обчислення (знизу-вгору), уникаючи необхідності багаторазового обходу синтаксичного дерева.

На рис. 3 продемонстровано приклад використання макроса msa-rewrite. У наведеному сценарії правило трансформації налаштовано на пошук операції додавання. Особливістю цього прикладу є те, що трансформація застосовується виключно до операції додавання верхнього рівня, тоді як її операнди (ліва та права частини) трактуються як цілісні піддерева без аналізу їхньої внутрішньої структури на цьому етапі.

Тобто, результатом переписування вхідного синтаксичного дерева, що є представленням виразу (1 + 2) + (3 + 2), буде нове дерево, що відповідає виразу ((1 + 2) + (3 + 2)) * 2. Це демонструє здатність системи маніпулювати складними вкладеними структурами, зберігаючи їх цілісність під час модифікації батьківських вузлів.

```
(fsa-use-lang :java)

(msa-rewrite tree [expression $expr1 $expr2]
 (:pre (:expression "$expr1 + $expr2"))
 (msa-parse-pattern :expression "( $expr1 + $expr2 ) * 2"))
```

Рис. 3. Структурна трансформація синтаксичного дерева
Fig. 3. Structural transformation of a syntax tree

Результати досліджень

Для практичної апробації та верифікації запропонованого підходу було реалізовано прикладну задачу статичного аналізу коду – побудову графу потоку виконання (CFG) для програм, написаних мовою Java.

Вибір цієї задачі зумовлений необхідністю обробки складних деревоподібних структур та розпізнавання різноманітних синтаксичних конструкцій, що є репрезентативним сценарієм для оцінки можливостей макросів msa-rewrite та msa-match.

При побудові CFG було розроблено набір синтаксичних шаблонів, що покривають ключові керуючі конструкції мови Java.

Специфікація правил трансформації включає:

1. оголошення методу:

(: *methodDeclaration* "$*typeOrVoid* $*identifier* $ *formalParameters* $*block*")

2. блоки коду:

(: *block* "$*block*")

(: *statement* "$*block*")

3. умовні оператори (розгалуження):

(: statement "if ($expression) $statement")

(: statement "if ($expression) $statement1 else $statement2")

4. циклічні конструкції (цикли з передумовою):

(: *statement* "*while* ($*expression*) $*statement*")

5. оператори повернення керування:

(: statement " return;")

(: statement " return $expression;")

Інші синтаксичні конструкції мови Java (вирази присвоювання, виклики методів тощо) у рамках даного експерименту розглядалися як лінійні атомарні інструкції, оскільки вони не формують специфічних розгалужень у графі потоку виконання.

Алгоритм роботи розробленого аналізатора базується на рекурсивному обході AST і складається з трьох етапів:

1) зіставлення поточного піддерева з одним із вищеописаних зразків;

2) у разі успішного зіставлення, у тілі правила (action) виконується створення відповідного вузла або ребра CFG;

3) для конструкцій, що містять вкладені блоки (наприклад, methodDeclaration або block), ініціюється рекурсивний обхід дочірніх вузлів для коректного з'єднання елементів графу;

Для оцінки ефективності запропонованого методу було проведено порівняльне дослідження з класичним імперативним підходом. Для цього було розроблено дві функціонально еквівалентні реалізації програми побудови CFG:

1. Експериментальна реалізація (DALT), де використовуються запропоновані макроси, квазіцитування та зіставлення зі зразком. Обсяг коду складає 141 лексичних одиниць.

2. Контрольна реалізація, яка написана в імперативному стилі з використанням низькорівневих операцій роботи з AST (явні перевірки типів, приведення типів, ручний обхід списків нащадків). Обсяг коду складає 651 лексичних одиниць.

Тестування обох реалізацій на тестовому наборі вихідних кодів Java показало повну тотожність вихідних даних. Згенеровані графи потоку виконання є структурно ідентичними, що підтверджує коректність методу.

Аналіз отриманих метрик демонструє, що використання запропонованих декларативних конструкцій дозволило скоротити лексичний обсяг коду на 78%. Такий суттєвий розрив пояснюється тим, що значна частина імперативного коду припадає на «інфраструктурну» логіку (навігація, перевірки, зв'язування змінних), яку DALT приховує всередині механізмів розгортання макросів. Навіть за умови винесення повторюваних операцій у допоміжні функції, імперативний підхід програє у лаконічності через необхідність явного опису кроків виконання, тоді як декларативний підхід описує лише бажану структуру. Це підтверджує гіпотезу про вищу виразність та ефективність запропонованих засобів метапрограмування для задач розробки інструментарію аналізу та трансформації синтаксичних дерев.

Висновки

У цій роботі представлено метод обробки синтаксичних дерев для підвищення ефективності розробки засобів аналізу та модифікації програмного коду на основі механізму квазіцитування, та спроєктована відповідна вбудована предметно-орієнтованої мови (eDSL) у середовищі Clojure для його реалізації.

Основними науковими та практичними результатами роботи є:

1. Запропоновано метод маніпулювання абстрактними синтаксичними деревами (AST). На відміну від класичних підходів, що оперують ієрархіями класів, запропонований метод базується на квазіцитуванні конкретного синтаксису. Це дозволило уніфікувати процеси аналізу та синтезу коду, використовуючи єдину нотацію для опису операндів-дерев – синтаксичні шаблони, що безпосередньо відображають структуру цільової мови.
2. Розроблено та формалізовано систему декларативних конструкцій. Описано набір макросів які забезпечують уніфікований інтерфейс для задач пошуку в AST, деструктуризації вузлів та генерації нових синтаксичних структур. Це дозволило підняти рівень абстракції при описі алгоритмів статичного аналізу.
3. Апробація методу на задачі побудови графів потоку виконання для мови Java показала, що використання DALT дозволяє істотно скоротити обсяг програмного коду порівняно з імперативною реалізацією. Декларативний опис усуває необхідність у ручній навігації по дереву та перевірці типів, що підвищує читабельність та надійність інструментарію.
4. Запропонований підхід вирішує проблему розриву між синтаксисом цільової мови та її внутрішнім представленням в аналізаторі, що значно спрощує розробку засобів рефакторингу та трансляції коду.

У даній статті основну увагу було зосереджено на описі методу, а також концептуальному апараті та семантиці запропонованої мови. Питання внутрішньої архітектури, алгоритмів зіставлення та оптимізації продуктивності будуть розглянуті у наступних публікаціях.

Перспективи подальших досліджень охоплюють розширення набору декларативних конструкцій, а також оптимізацію продуктивності механізмів зіставлення та переписування для роботи з великими синтаксичними деревами.

Список використаних джерел

1. Hickey R. A history of Clojure. Proceedings of the ACM on Programming Languages. 2020. Vol. 4, HOPL. P. 1–46. URL: https://doi.org/10.1145/3386321. Дата звернення: 16.12.2025.
2. Генератор синтаксичних аналізаторів ANTLR. Режим доступу: https://www.antlr.org. Дата звернення: 16.12.2025.
3. Kosar T., Bohra S., Mernik M. Domain-Specific Languages: A Systematic Mapping Study. Information and Software Technology. 2016. Vol. 71. P. 77–91. URL: https://doi.org/10.1016/j.infsof.2015.11.001 Дата звернення: 16.12.2025.
4. Cordy J. R. TXL - A Language for Programming Language Tools and Applications. Electronic Notes in Theoretical Computer Science. 2004. Vol. 110. P. 3–31. URL: https://doi.org/10.1016/j.entcs.2004.11.006. Дата звернення: 16.12.2025.
5. OIL: an industrial case study in language engineering with Spoofax / O. Bunte et al. Software and Systems Modeling. 2024. URL: https://doi.org/10.1007/s10270-024-01185-x. Дата звернення: 16.12.2025.
6. Large-scale semi-automated migration of legacy C/C++ test code / M. T. W. Schuts et al. Software: Practice and Experience. 2022. URL: https://doi.org/10.1002/spe.3082. Дата звернення: 16.12.2025.
7. Tom: Piggybacking Rewriting on Java / E. Balland et al. Lecture Notes in Computer Science. Berlin, Heidelberg. P. 36–47. URL: https://doi.org/10.1007/978-3-540-73449-9_5. Дата звернення: 16.12.2025.
8. Sloane A. M., Roberts M. Oberon-0 in Kiama. Science of Computer Programming. 2015. Vol. 114. P. 20–32. URL: https://doi.org/10.1016/j.scico.2015.10.010. Дата звернення: 16.12.2025.
9. Касянчук Д., Марченко О. Засіб синтаксичного розбору на основі генератора синтаксичних аналізаторів ANTLR та мови Clojure. КОМП'ЮТЕРНО-ІНТЕГРОВАНІ ТЕХНОЛОГІЇ: ОСВІТА, НАУКА, ВИРОБНИЦТВО. 2024. № 56. С. 174–184. URL: https://doi.org/10.36910/6775-2524-0560-2024-56-22. Дата звернення: 16.12.2025.

References

1. R. Hickey, "A history of Clojure," Proc. ACM Program. Lang., vol. 4, HOPL, pp. 1–46, Jun. 2020. Accessed: Dec. 16, 2025. [Online]. Available: https://doi.org/10.1145/3386321
2. ANTLR parser generator. Accessed: Dec. 16, 2025. Available: https://www.antlr.org.
3. T. Kosar, S. Bohra, and M. Mernik, "Domain-Specific Languages: A Systematic Mapping Study," Inf. Softw. Technol., vol. 71, pp. 77–91, Mar. 2016. Accessed: Dec. 16, 2025. [Online]. Available: https://doi.org/10.1016/j.infsof.2015.11.001
4. J. R. Cordy, "TXL - A Language for Programming Language Tools and Applications," Electron. Notes Theor. Comput. Sci., vol. 110, pp. 3–31, Dec. 2004. Accessed: Dec. 16, 2025. [Online]. Available: https://doi.org/10.1016/j.entcs.2004.11.006
5. O. Bunte et al., "OIL: an industrial case study in language engineering with Spoofax," Softw. Syst. Model., Jun. 2024. Accessed: Dec. 16, 2025. [Online]. Available: https://doi.org/10.1007/s10270-024-01185-x
6. M. T. W. Schuts, R. T. A. Aarssen, P. M. Tielemans, and J. J. Vinju, "Large-scale semi-automated migration of legacy C/C++ test code," Softw.: Pract. Experience, Mar. 2022. Accessed: Dec. 16, 2025. [Online]. Available: https://doi.org/10.1002/spe.3082
7. E. Balland, P. Brauner, R. Kopetz, P.-E. Moreau, and A. Reilles, "Tom: Piggybacking Rewriting on Java," in Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berl. Heidelb., n.d., pp. 36–47. Accessed: Dec. 16, 2025. [Online]. Available: https://doi.org/10.1007/978-3-540-73449-9_5
8. A. M. Sloane and M. Roberts, "Oberon-0 in Kiama," Sci. Comput. Program., vol. 114, pp. 20–32, Dec. 2015. Accessed: Dec. 16, 2025. [Online]. Available: https://doi.org/10.1016/j.scico.2015.10.010
9. D. Kasianchuk and O. Marchenko, " A parsing tool based on the ANTLR parser generator and the Clojure language" COMPUTER-INTEGR. TECHNOL.: EDUC., SCI., PROD., no. 56, pp. 174–184, Sep. 2024. Accessed: Dec. 16, 2025. [Online]. Available: https://doi.org/10.36910/6775-2524-0560-2024-56-22

Отримана в редакції 05.12.2025. Прийнята до друку 15.12.2026. Розміщено в інтернеті 30 березня 2026.
Received 05 December 2025. Approved 15 December 2026. Available in Internet 30 March 2026