DOI: https://doi.org/10.36910/6775-2524-0560-2024-56-22
УДК 004.4`2
Касянчук Дмитро Павлович, аспірант
https://orcid.org/0009-0004-2824-8232
Марченко Олександр Іванович, к.т.н., доцент
https://orcid.org/0000-0002-4537-3420
Національний технічний університет України «Київський політехнічний інститут імені Ігоря Сікорського», м. Київ, Україна

ЗАСІБ СИНТАКСИЧНОГО РОЗБОРУ НА ОСНОВІ ГЕНЕРАТОРА СИНТАКСИЧНИХ АНАЛІЗАТОРІВ ANTLR ТА МОВИ CLOJURE

Касянчук Д.П., Марченко О.І. Засіб синтаксичного розбору на основі генератора синтаксичних аналізаторів ANTLR та мови Clojure. Ця стаття присвячена процесу синтаксичного розбору програмного коду. В результаті виконаного аналізу низки публікацій, які містять порівняння існуючих генераторів синтаксичних аналізаторів, було визначено, що генератор синтаксичних аналізаторів ANTLR користується популярністю у розробників завдяки своїй простоті, широкому спектру функцій, а також якості і ефективності згенерованого коду. В статті представлено засіб синтаксичного розбору програмного коду, що базується на використанні генератора синтаксичних аналізаторів ANTLR, а також сучасної функційної мови програмування Clojure, характеристики якої дозволяють значно легше створювати деревоподібні структури даних, а також аналізувати та змінювати їх, якщо порівнювати з нефункційними мовами програмування. Крім того, широкий спектр конструкцій, які має мова Clojure, дозволяють робити код лаконічнішим та зрозумілішим, ніж код записаний нефункційними мовами програмування. Це є великою перевагою при обробці складних структур даних, таких як синтаксичні дерева. На відміну від існуючих бібліотек мови Clojure, що надають деякі засоби синтаксичного розбору, які використовують генератор ANTLR у неефективному режимі інтерпретатора, розроблений засіб містить функції, які надають розробнику повноцінні можливості виконання синтаксичного розбору програмного коду за допомогою ANTLR. Множина цих функцій включає: функцію генерації класів лексичного та синтаксичного аналізаторів, яка має використовуватися в якості окремого етапу збірки проєкту; функцію реєстрації, мета якої полягає у аналізі класів згенерованих аналізаторів і створенні структур даних мови Clojure для реалізації інтерфейсу взаємодії між програмами, написаними мовою Clojure і згенерованими класами аналізаторів, написаними мовою Java; функцію синтаксичного розбору програмного коду, що використовує зареєстровані структури даних. Завдяки тому, що програмний код, написаний мовою Clojure, працює на основі Java Virtual Machine (JVM), в цих функціях вдалося використати класи бібліотек генератора ANTLR, що написані мовою Java.
Ключові слова: ANTLR, Clojure, синтаксичний аналіз, синтаксичне дерево, лексичний аналіз, токен, рефлексія

Kasianchuk D., Marchenko O. A parsing tool based on the ANTLR parser generator and the Clojure language. This article is devoted to the process of parsing program code. As a result of the analysis of a number of publications comparing existing parser generators, it has been determined that the ANTLR parser generator is popular among developers due to its simplicity, wide range of functions, as well as the quality and efficiency of the generated code. The article presents a tool for parsing program code based on the use of the ANTLR parser generator and the modern functional programming language Clojure, whose characteristics make it much easier to create tree-like data structures, as well as analyze and modify them, when compared to non-functional programming languages. In addition, the wide range of constructs available in Clojure makes it possible to make code more concise and understandable than code written in non-functional programming languages. This is a great advantage when processing complex data structures such as syntax trees. Unlike the existing Clojure libraries that provide some parsing tools that use the ANTLR generator in an inefficient interpreter mode, the developed tool contains functions that provide the developer with full-fledged capabilities to perform parsing of program code using ANTLR. The set of these functions includes: the function of generating classes of lexical and syntax analyzers, which should be used as a separate stage of project build; the registration function, the purpose of which is to analyze the classes of generated analyzers and create Clojure data structures to implement an interface between programs written in Clojure and generated classes of analyzers written in Java; the function of parsing program code using registered data structures. Due to the fact that the program code written in Clojure runs on the Java Virtual Machine (JVM), these functions were able to use the classes of the ANTLR generator libraries written in Java.
Keywords: ANTLR, Clojure, parsing, parse tree, lexical analysis, token, reflection

Постановка наукової проблеми

Основою засобів статичного аналізу програмного коду є спеціальне внутрішнє представлення цього коду, зазвичай, синтаксичне дерево, яке є результатом синтаксичного розбору рядка термінальних символів, що відповідає правилам формальної граматики. Така граматика може описувати, як природну мову (напр. українську або англійську), так і мову програмування або формат даних. Процес виділення лексем (або токенів) з початкового програмного коду, називають лексичним аналізом. Синтаксичні аналізатори – це програмні засоби, які на основі результату лексичного аналізу, перевіряють вхідний текст на відповідність синтаксису та будують синтаксичне дерево, що представляє вхідну програму деревоподібною структурою даних. Хоча певний аналіз вхідного коду програми можна зробити і в оригінальній текстовій формі, використання ієрархічної деревовидної структури даних для представлення цього коду є набагато ефективнішим і зручнішим,

© Касянчук Д.П., Марченко О.І.

ніж виконання перетворень безпосередньо початкового текстового представлення програмного коду.

З вищесказаного можна зробити висновок, що розробка нових ефективних і зручних для використання засобів обробки тексту на основі синтаксичних дерев є актуальною.

Аналіз досліджень

Широке використання синтаксичного аналізу в сфері програмного забезпечення породжує попит у створенні засобів, мета яких полягає у автоматичному створенні лексичних та синтаксичних аналізаторів. В таких засобах розробник описує синтаксис мови, якою написаний програмний код, що має бути розпізнаним, а в результаті отримує аналізатори для заданої граматики.

На сьогоднішній день, існує багато генераторів синтаксичних аналізаторів, які також називають ще компіляторами компіляторів (compiler-compiler), наприклад:

1) ANTLR [1];
2) Lex/Yacc [2];
3) LISA [3].

В роботі [4] дослідники здійснили якісне порівняння генераторів синтаксичних аналізаторів LISA , Lex/Yacc та ANTLR 3. Для виконання поставленої задачі вони визначили граматику предметно-орієнтованої мови Lavanda, метою якої є опис мішків для білизни, які компанія відправляє на прання.

Якісну оцінку цих трьох генераторів синтаксичних аналізаторів автори виконують на основі наступних критеріїв:

1) специфікація мови;
2) процес генерації синтаксичних аналізаторів;
3) згенерований код.

Дослідники стверджують, що за показниками зручності використання, наявності додаткових засобів та якості кінцевого продукту Lex/Yacc є найбіднішим і застарілим засобом, в той час як інші два генератори є достатньо ефективними, а також дуже схожими між собою.

Незважаючи на те, що ці засоби відрізняються такими своїми властивостями як стратегія синтаксичного аналізу або мова програмування згенерованих аналізаторів, переважна більшість розробників обирають або класичний засіб Lex/Yacc, або засіб ANTLR.

В роботі [5] здійснено емпіричне дослідження, метою якого є аналіз успішності студентів, що розроблюють компілятор, в залежності від використання ними засобів Lex/Yacc або ANTLR. Дослідники резюмують, що генератор ANTLR значно переважає Lex/Yacc. Студенти, які використовували ANTLR, показали значно кращі результати при виконанні як початкових завдань з розробки лексичного та синтаксичного аналізаторів, так і завдань з подальших модифікацій цих аналізаторів. Також ці студенти мали вищі показники за кількістю виконаних завдань, окремих оцінок за лабораторні роботи, а також підсумкових оцінок, порівняно зі студентами, що виконували розробку з використанням Lex/Yacc. За відгуками студентів ANTLR є більш інтуїтивним та простим засобом.

В якості мови програмування для розробки нового засобу для підтримки процесів лексичного і синтаксичного аналізу, що пропонується в цій статті, був обраний сучасний діалект мови Lisp – Clojure [6]. Проаналізуємо властивості цієї мови та її перспективність для роботи з ієрархічними структурами даних.

Оскільки мова Clojure є функційною мовою програмування, то її основними властивостями є, в першу чергу, загальні властивості функційних мов:

1) незмінність даних (варто відзначити, що в мові Clojure також існує багато засобів для реалізації змінного стану);

2) функції також є структурами даних нарівні з іншими структурами даних (тобто, вони можуть передаватися в інші функції, як аргументи, або повертатися з них, як результат);

3) функції вищого порядку;
4) чисті функції;
5) спискові структури даних.

Такі властивості дозволяють значно легше створювати деревоподібні структури даних, а також аналізувати та змінювати їх, якщо порівнювати з нефункційними мовами програмування. Крім того, широкий спектр конструкцій, які має мова Clojure, дозволяють робити код коротшим та зрозумілішим, що є великою перевагою при обробці складних структур даних, таких як синтаксичні дерева. Мова Clojure має широкий набір конструкцій для реалізації паралельних та розподілених обчислень. Ці конструкції можуть бути використанні у певних задачах статичного аналізу коду, які

© Касянчук Д.П., Марченко О.І.

можливо розпаралелити, з метою покращення продуктивності.

Програмний код, написаний мовою Clojure, працює на основі віртуальної машини Java (Java Virtual Machine – JVM), що дозволяє використовувати широкий спектр бібліотек, які існують у середовищі JVM [7]. Яскравим прикладом цього є використання класів бібліотек генератора ANTLR, а також класів лексичних та синтаксичних аналізаторів, згенерованих за його допомогою. Варто також зазначити, що ця мова програмування має систему макросів, що дозволяє додавати нові високорівневі синтаксичні конструкції, які орієнтовані на певні прикладні задачі.

Мова Clojure підтримує, так-званий, інтерактивний процес розробки, коли взаємодія з програмою відбувається вже під час її написання. Робота при такому процесі розробки здійснюється через Read-Eval-Print Loop (REPL), що дозволяє швидко визначати і тестувати різні функції без необхідності перекомпіляції всього вихідного коду проєкту.

Розглянемо дві бібліотеки мови Clojure для розробки лексичних і синтаксичних аналізаторів: clj-antlr [8] та instaparse [9].

Бібліотека clj-antlr місить функції для синтаксичного аналізу програмного коду з використанням генератора ANTLR у режимі інтерпретатора. Цей режим дозволяє спростити інтеграцію ANTLR у власний проєкт завдяки тому, що не потрібно виконувати етапи генерації лексичного і синтаксичного аналізаторів, а також компіляції згенерованого коду. В документації ANTLR зазначено, що такий режим зручно використовувати для невеликих завдань розбору програмного коду. Розглянемо переваги та недоліки режиму інтерпретатора.

Переваги:

1) цей режим дозволяє швидше виконувати розробку і тестування граматики (розробник може без затримки бачити результати змін, які він вносить у граматику, без необхідності генерування синтаксичного аналізатора і компілювання його коду);

2) простота інтеграції генератора ANTLR у проєкт.

Недоліки:

1) інтерпретатор працює не так швидко, як згенеровані засоби, особливо для вхідних програм великого розміру або для складних граматик;

2) деякі варіанти оптимізації вихідного коду можуть бути недоступними в цьому режимі.

Незважаючи на те, що розробники clj-antlr використовують ANTLR у неефективному режимі інтерпретатора для виконання задачі синтаксичного розбору, пропонований засіб, за їх словами, працює швидше, ніж засоби іншої бібліотеки instaparse, що демонструється ними на прикладі синтаксичного розбору JSON документу.

Розглянемо тепер більш детально бібліотеку instaparse.

Почнемо з головної відмінності, яка полягає у тому, що при використанні цієї бібліотеки для виконання процесу синтаксичного аналізу не потрібне використання ні генератора ANTLR, ні інших генераторів, а генерування синтаксичного аналізатора за заданою граматикою відбувається за допомогою функцій, написаних мовою Clojure. Розглянемо переваги та недоліки бібліотеки instaparse.

Переваги:

1) може бути використана для будь-якого типу граматики: ліво-рекурсивна, право-рекурсивна або двозначна;

2) для представлення вихідного дерева використовуються найпопулярніші формати мови Clojure, такі як вектор або хеш-таблиця;

3) має детальне звітування про помилки;

4) легке впровадження у проєкт.

Недоліки:

1) для великої кількості вхідних даних або у випадку складної граматики, процес синтаксичного розбору з використанням бібліотеки instaparse може виконуватися повільніше порівняно з синтаксичними аналізаторами, написаними вручну або згенерованими спеціалізованими засобами;

2) звіт про помилки може бути занадто перевантаженим, що ускладнює виявлення проблем у складних граматиках;

3) на відміну від ANTLR, не має напрацьованої бази готових граматик для великої кількості мов програмування [10].

Підсумовуючи вище описане, обидві бібліотеки мови Clojure містять функції для підтримки процесів лексичного і синтаксичного аналізу, які є ефективними тільки для нескладних граматик, а

© Касянчук Д.П., Марченко О.І.

також при аналізі невеликих програм. Переваги цих бібліотек полягають, швидше, у надані зручного інтерфейсу для розробника, і легкій інтеграції засобів синтаксичного розбору у проєкт, а не у швидкодії виконання процесів лексичного і синтаксичного аналізу.

Постановка завдання

Метою цієї статті є розробка нового ефективного засобу реалізації синтаксичного розбору вхідної програми, призначеного для розробників, які, з однієї сторони хочуть використовувати функційну мову програмування Clojure, а з другої сторони – популярний генератор синтаксичних аналізаторів ANTLR, який написаний мовою Java. Основною задачею розробки такого засобу є реалізація інтерфейсу взаємодії між програмами, написаними мовою Clojure, і засобами генератора ANTLR, написаними мовою Java.

Виклад основного матеріалу

Розроблений авторами засіб, який пропонується в цій статті, містить наступні функції, що написані мовою Clojure:

1) функція antlr4, призначення якої полягає у генерації коду лексичного і синтаксичного аналізаторів;
2) функція register-lang, призначення якої полягає у створені структур даних мови Clojure, які реалізують інтерфейс взаємодії між кодом програм, написаних мовою Clojure, і кодом згенерованих аналізаторів, написаними мовою Java;
3) функція parse-source, яка використовує зареєстровані структури даних і призначення якої полягає у виконанні власне синтаксичного розбору вхідного програмного коду.

Ці функції реалізують доступ до класів та методів генератора синтаксичних аналізаторів ANTLR. Розглянемо більш детально ці функції. В якості засобу для збірки та розповсюдження розробленого проєкту був обраний відомий засіб Leiningen [11]. Це один з найпоширеніших засобів для розробки проєктів мовою Clojure, який здобув свою популярність завдяки простоті у використанні, а також широкому спектру можливостей, які він надає. Конфігурація проєкту визначається у спеціальному файлі – project.clj, приклад якого зображено на рис.1.

```
(defproject <project-name> <version>
  :plugins [[lein-pprint "1.1.1"] ...]
  :source-paths ["src"]
  :java-source-paths ["src/java"]
  :prep-tasks ["javac" "compile"]
  :dependencies [[org.clojure/clojure "1.11.1"] ...]
  ...)
```

Рис. 1. Приклад project.clj файлу

Цей файл містить опис налаштувань, які необхідні для роботи функцій розробленого засобу:

1. :plugins – містить список плагінів. Кожен плагін містить функції, які мають доступ до налаштувань файлу project.clj. Ці функції можуть бути використані під час збірки проєкту, як додаткові етапи.
2. :source-paths – список, кожен елемент якого є відносним шляхом до директорії, яка містить файли з програмним кодом, що написаний мовою Clojure.
3. :java-source-paths – список, кожен елемент якого є відносним шляхом до директорії, яка містить файли з програмним кодом, що написаний мовою Java.
4. :prep-tasks – список етапів збірки проєкту, в який можна додати функції з п. 1. За замовчуванням, цей список містить наступні етапи:
   1) javac – виконує компіляцію файлів, що знаходяться в java-source-paths (п. 3);
   2) compile – виконує компіляцію файлів, що знаходяться в source-paths (п. 2).
5. :dependencies – список залежностей проєкту. Містить назви бібліотек та їх версії, які автоматично завантажуються з центрального репозиторію Maven або інших джерел.

Розглянемо детально функцію antlr4, з якої починається інтеграція генератора ANTLR у проєкт, що розроблюється. Мета цієї функції полягає у генерації коду лексичних і синтаксичних аналізаторів для певної вхідної мови. Функція antlr4 була розроблена і реалізована в рамках окремого плагіну засобу Leiningen. Цей плагін отримав ім'я lein-clj-parsing. Це було зроблено для того, щоб автоматизувати процес генерації коду аналізаторів шляхом додавання цієї функції у етапи збірки проєкту.

© Касянчук Д.П., Марченко О.І.

Параметри функції antlr4:

1) project – конфігурація project.clj файлу, яка має містити налаштування, необхідні для генерації коду аналізаторів; ці налаштування представлені хеш-таблицею, що пов’язана з ключовим словом :antlr, і має містити наступні ключі:

a) :grammars-dir – шлях до директорії, піддиректорії якої містять необхідні для генерації коду лексичного та синтаксичного аналізаторів файли, а ім’я кожної піддиректорії відповідає назві граматики;

b) :package – ім’я Java пакету, в якому будуть знаходитися згенеровані класи; варто зазначити, що це ім’я доповнюється іменем граматики;

c) :out-dir – шлях до директорії, яка міститиме згенеровані класи, що написані мовою Java;

2) grammars – (необов’язковий) список імен граматик (тобто відповідних директорій), для яких треба згенерувати аналізатори; за замовчуванням, аналізатори генеруються для всіх наявних граматик.

В список залежностей розробленого плагіну була додана бібліотека org.antlr/antlr4 генератора ANTLR, що містить клас Tool, який призначений для генерації коду лексичного і синтаксичного аналізаторів. Також, варто відзначити, що генерація аналізаторів для певної граматики відбувається тільки у випадку відсутніх згенерованих Java файлів або якщо відбулися зміни у директорії граматики. Перевірка таких змін базується на порівнянні часу останньої зміни директорії граматики і директорії зі згенерованими Java файлами для цієї граматики.

Для використання функції antlr4 у проєкті розробник має зробити наступні налаштування у файлі project.clj:

1) додати lein-clj-parsing у список плагінів (:plugins);
2) додати налаштування, що необхідні для генерації коду аналізаторів (:antlr);
3) додати функцію antlr4 у список етапів збірки проєкту перед javac етапом (:prep-tasks);
4) додати значення :out-dir налаштування у список Java директорій (:java-source-paths) для того, щоб згенеровані аналізатори компілювалися під час збірки проєкту.

Приклад project.clj файлу з описаними вище налаштуваннями, зображено на рис. 2.

```
(defproject <project-name> <version>
  :plugins [[lein-clj-parsing "0.1.0"] ...]
  :antlr {:grammars-dir "resources/grammars"
          :out-dir "src/java"
          :package "parser.generated"}
  :prep-tasks ["antlr4" "javac" "compile"]
  :java-source-paths ["src/java"]
  ...)
```

Рис. 2. Приклад project.clj файлу з налаштуваннями для генерації коду аналізаторів

Функції запропонованого авторами засобу, що описуються далі, були розроблені і реалізовані в рамках окремого проєкту засобу Leiningen, що отримав ім’я clj-parsing. Тобто, для їх використання, розробник має додати clj-parsing в список залежностей проєкту (:dependencies).

В список залежностей проєкту clj-parsing була також додана бібліотека org.antlr/antlr4-runtime генератора ANTLR, оскільки вона містить функції, що необхідні для створення і обробки синтаксичних дерев, отриманих за допомогою згенерованих аналізаторів.

Розглянемо основні функції проєкту clj-parsing.

Після виконання генерації, створені класи аналізаторів мають бути обов’язково зареєстровані за допомогою функції register-lang. Суть реєстрації полягає у аналізі класів згенерованих аналізаторів за допомогою Java рефлексії [12] і у створені структур даних мови Clojure, які реалізують інтерфейс взаємодії між кодом програм, написаних мовою Clojure, і кодом згенерованих аналізаторів, написаних мовою Java. Рефлексія допомагає досліджувати вміст класу, а також викликати його методи, незалежно від специфікатора доступу, який з ними використовується.

Опишемо структури даних мови Clojure, які будуть створені в результаті роботи функції register-lang (нагадаємо, що у функційних мовах функції також є структурами даних):

1. Функція make-lexer. Ця функція в якості параметра приймає потік символів вхідної програми, для якої буде виконуватися лексичний аналіз. Цей потік символів мусить мати

© Касянчук Д.П., Марченко О.І.

спеціальний тип CharStream з бібліотеки генератора ANTLR. Функція make-lexer повертає об'єкт лексичного аналізатора для потоку символів певної вхідної програми.

2. Функція make-parser. Ця функція через параметр приймає потік токенів, який є результатом роботи лексичного аналізатора і для якого буде виконаний синтаксичний аналіз. Цей потік має бути об'єктом класу, що реалізує інтерфейс TokenStream з бібліотеки генератора ANTLR. Функція make-parser повертає об'єкт синтаксичного аналізатора для конкретного потоку токенів перетвореної вхідної програми.

3. Хеш-таблиця rule-parse-fn. Ключем цієї таблиці є ім'я правила граматики, а значенням – функція, яка в якості параметра приймає об'єкт синтаксичного аналізатора і викликає його метод. Цей метод синтаксичного аналізатора повертає дерево, яке є результатом синтаксичного розбору, що починається з правила, з ім'ям якого ця функція пов'язана у хеш-таблиці.

Параметри функції register-lang:

1) lang-name – ім'я граматики, з яким будуть пов'язані створені структури даних;
2) lexer-class – згенерований клас лексичного аналізатора;
3) parser-class – згенерований клас синтаксичного аналізатора.

На рис. 3 зображено реалізацію функції реєстрації register-lang. В цій функції описані вище структури даних зберігаються у глобальній хеш-таблиці languages, в якій ключем є значення параметра lang-name (тобто, імені граматики), а значенням є вкладена хеш-таблиця з наступними ключами:

1) :make-lexer, що пов'язується з функцією make-lexer;
2) :make-parser, що пов'язується з функцією make-parser;
3) :rule-parse-fn, що пов'язується з хеш-таблицею rule-parse-fn.

```
(defonce languages (atom {}))

(defn register-lang [lang-name ^Class lexer-class ^Class parser-class]
  (let [lang-name (name lang-name)]
    (check-lang-input lang-name lexer-class parser-class)
    (let [lexer-cnstr (.getDeclaredConstructor lexer-class (class-array [CharStream]))
          parser-cnstr (.getDeclaredConstructor parser-class (class-array [TokenStream]))
          rule-names (-> (.getField parser-class "ruleNames") (.get nil))]
      (->>
       (for [rule-name rule-names
             :let [method (.getMethod parser-class rule-name (class-array []))]]
         [rule-name (fn [parser] (.invoke method parser (object-array [])))])
       (into {})
       (hash-map :make-lexer (fn [stream] (.newInstance lexer-cnstr (object-array [stream])))
                 :make-parser (fn [tokens] (.newInstance parser-cnstr (object-array [tokens])))
                 :rule-parse-fn)
       (swap! languages assoc lang-name)))))
```

Рис. 3. Програмна реалізація функції реєстрації register-lang

Оскільки код вхідної програми може зберігатися у різних структурах даних (файл, потік, рядок тощо), а функція створення об'єкта лексичного аналізатора приймає в якості параметра тільки потік символів (див. register-lang), то для формування потоку символів з різних структур даних було вирішено створити протокол мови Clojure (протокол – це набір іменованих методів з сигнатурами без конкретної реалізації). У цьому протоколі, що отримав ідентифікатор CharStreamBuilder, був реалізований тільки один метод make-stream, який власне і виконує таке формування, приймаючи в якості параметра структуру даних з кодом вхідної програми і формує із символів цього коду потік символів. Існуючі реалізації методу make-stream використовують методи класу CharStreams з бібліотеки генератора ANTLR, призначення яких полягає у формуванні потоку символів із символів вхідної програми, яка може знаходитися в наступних структурах даних (рис. 4): 1) файл; 2) потік символів; 3) потік байтів; 4) рядок.

```
(defprotocol CharStreamBuilder
  (^CharStream make-stream [source] "Create CharStream from `source`"))

(extend-protocol CharStreamBuilder
  File
  (make-stream [source] (CharStreams/fromFileName (.getPath source)))
  Path
  (make-stream [source] (CharStreams/fromPath source))
  Reader
  (make-stream [source] (CharStreams/fromReader source))
  InputStream
  (make-stream [source] (CharStreams/fromStream source))
  String
  (make-stream [source] (CharStreams/fromString source)))
```

Рис. 4. Протокол переписування програмного коду із заданої початкової структури даних у потік символів

При необхідності, метод make-stream може бути легко розширений реалізаціями для інших структур даних.

Функція parse-source створює об'єкти лексичного і синтаксичного аналізаторів і виконує власне синтаксичний розбір програмного коду.

Параметри функції parse-source:

1) lang-name – ім'я зареєстрованої граматики;
2) rule – ім'я правила граматики, з якого треба починати розбір програмного коду;
3) source – структура даних, що містить програмний код.

На рис. 5 схематично зображено алгоритм роботи цієї функції, а також використання зареєстрованих структур даних і методу створеного протоколу CharStreamBuilder.

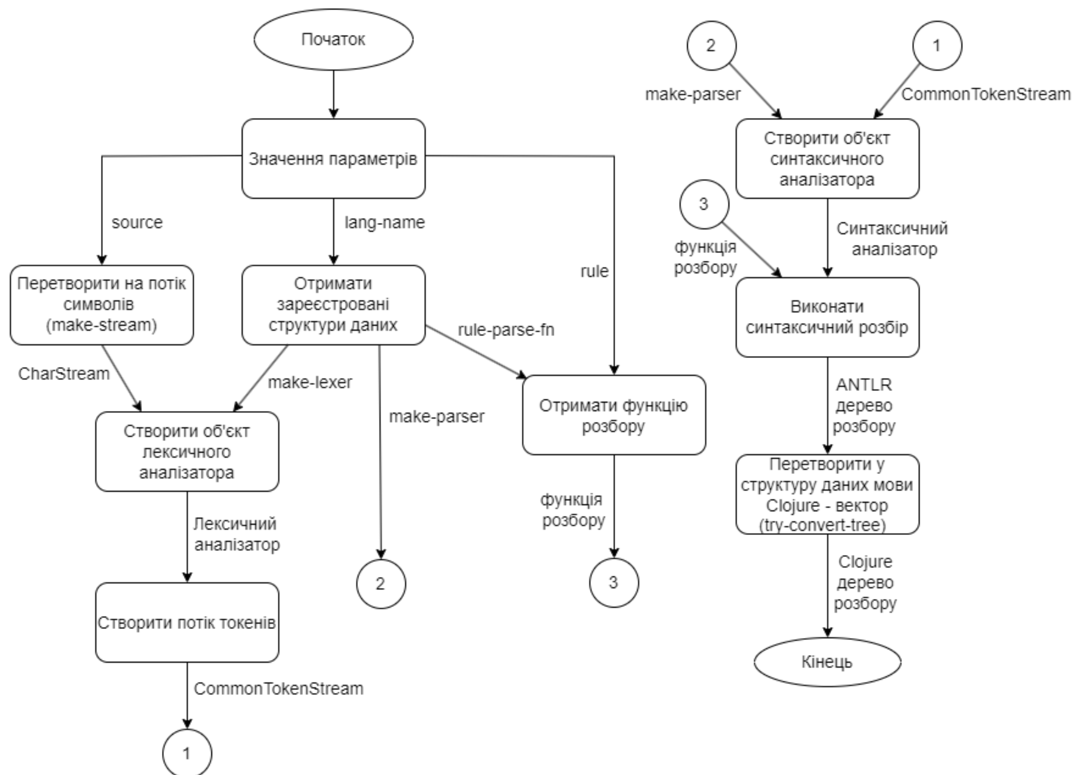

Рис. 5. Алгоритм роботи функції розбору parse-source

На рис. 6 зображено програмну реалізацію функції parse-source. В якості класу для створення потоку токенів був обраний клас CommonTokenStream з бібліотеки генератора ANTLR. Цей клас використовується у більшості випадків для отримання потоку токенів. Для створення об'єкта класу CommonTokenStream, в його конструктор необхідно передати об'єкт лексичного аналізатора в якості

аргументу.

```
(defn parse-source [lang-name rule source]
  (if-let [{:keys [make-lexer make-parser rule-parse-fn]} (@languages lang-name)]
    (let [lexer (-> source make-stream make-lexer)
          tokens (CommonTokenStream. lexer)
          parser (make-parser tokens)]
      (if-let [parse-fn (rule-parse-fn rule)]
        (try-convert-tree (parse-fn parser) parser)
        (throw (IllegalArgumentException.
                (str "Invalid rule name - " rule ".")))))
    (throw (IllegalArgumentException.
             (str "Language '" lang-name "' isn't registered.")))))
```

Рис. 6. Функція parse-source

Функція try-convert-tree перетворює синтаксичне дерево, що представлене об'єктом класу ParserRuleContext з бібліотеки генератора ANTLR, у синтаксичне дерево, що представлене структурою даних «вектор», яка є у мові Clojure.

Параметри функції try-convert-tree:

1) node – початкова вершина (корінь) дерева ANTLR;
2) parser – об'єкт синтаксичного аналізатора.

Перетворення відбувається тільки у випадку, якщо:

- не знайдено жодної синтаксичної помилки;
- процес розбору використав всі наявні токени; ця перевірка була додана через те, що синтаксичний аналізатор, згенерований за допомогою ANTLR, вважає розбір успішним, незважаючи на токени, які залишилися нерозібраними; ця перевірка виконується шляхом перевірки наступного (у потоці токенів) токену на рівність спеціальному значенню, що представляє кінець тексту.

Опишемо більш детально алгоритм перетворення дерева у форматі ANTLR у формат типу «вектор» мови Clojure.

У випадку, якщо вершина є термінальною (TerminalNode), то об'єкт токена цієї вершини перетворюється у хеш-таблицю, що містить значення типу і тексту цього токена, які пов'язуються з ключовими словами :type і :text відповідно.

Наприклад, термінальна вершина, яка зображена у вигляді UML діаграми об'єктів, на рис. 7, перетворюється у хеш-таблицю, зображену на рис. 8.

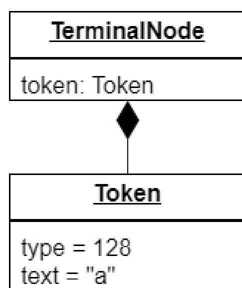

Рис. 7. Термінальна вершина

```
{:type 128
 :text "a"}
```

Рис. 8. Перетворена термінальна вершина

У випадку, якщо вершина є нетермінальною (ParserRuleContext), то створюється вектор. Першим його елементом є ім'я правила граматики, яке отримується за допомогою функції, зображеної на рис. 9.

```
(defn- rule-name [^Parser parser ^ParserRuleContext node]
  (->> node .getRuleIndex (get (.getRuleNames parser)) keyword))
```

Рис. 9. Функція отримання імені правила граматики з нетермінальної вершини

Решта елементів вектора – це результати перетворення дочірніх вершин цієї нетермінальної вершини. Наприклад, нетермінальна вершина з дочірніми вершинами, які зображені на рис. 10,

© Касянчук Д.П., Марченко О.І.

перетворюється у вектор, зображений на рис. 11.

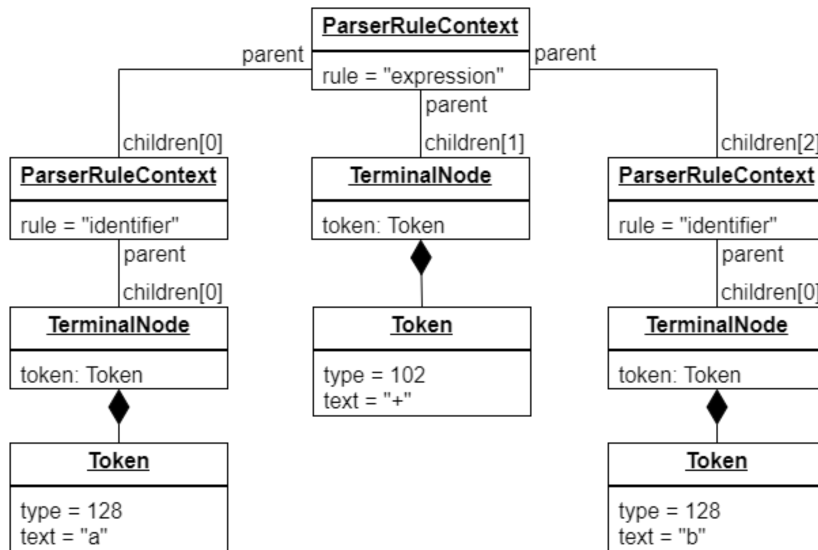

Рис. 10. Нетермінальна вершина

```
[:expression
 [:identifier {:type 128, :text "a"}]
 {:type 102, :text "+"}
 [:identifier {:type 128, :text "b"}]]
```

Рис. 11. Перетворена нетермінальна вершина

На рис. 12 зображено програмну реалізацію функції try-convert-tree, яка виконує описане перетворення.

```
(defn try-convert-tree [node ^Parser parser]
  (letfn [(helper [node]
            (condp instance? node
              TerminalNode
              (let [token (.getSymbol node)]
                {:type (.getType token)
                 :text (.getText token)})
              ParserRuleContext
              (into [(rule-name parser node)] (map helper) (get-children node))
              :else (throw (IllegalArgumentException.
                             (str "Unrecognized tree type - " (class node) ".")))))]
    (let [errors-count (.getNumberOfSyntaxErrors parser)]
      (cond
        (pos? errors-count)
        (throw (IllegalStateException.
                (str "Parsing has failed. Number of errors - " errors-count ".")))
        (= (.. parser getInputStream (LA 1)) (Token/EOF))
        (helper node)
        :else (throw
               (IllegalStateException.
                (str "Rule '" (name (rule-name parser node)) "' hasn't consumed all input."))))))))
```

Рис. 12. Функція try-convert-tree

На рис. 13 зображено виклик функції parse-source для випадку виконання синтаксичного розбору програмного коду, написаного мовою Java. Розбір цього програмного коду починається з правила граматики, яке має ім’я statement. Також, припускаємо, що граматика java, була попередньо зареєстрована за допомогою функції register-lang, як показано на рис. 14.

```
(parse-source :java :statement "a = b;")
```

Рис. 13. Виклик функції parse-source

```
(register-lang :java
               parser.generated.java.JavaLexer
               parser.generated.java.JavaParser)
```

Рис. 14. Реєстрація граматики java за допомогою функції register-lang

Розглянемо приклад перетворення синтаксичного дерева з формату бібліотеки генератора ANTLR у формат типу «вектор» мови Clojure. На рис. 15 показана UML діаграма об'єктів синтаксичного дерева, яке було отримане за допомогою синтаксичного аналізатора, що згенерований за допомогою ANTLR, а на рис. 16 показане те ж саме дерево після його перетворення функцією try-convert-tree у вектор мови Clojure.

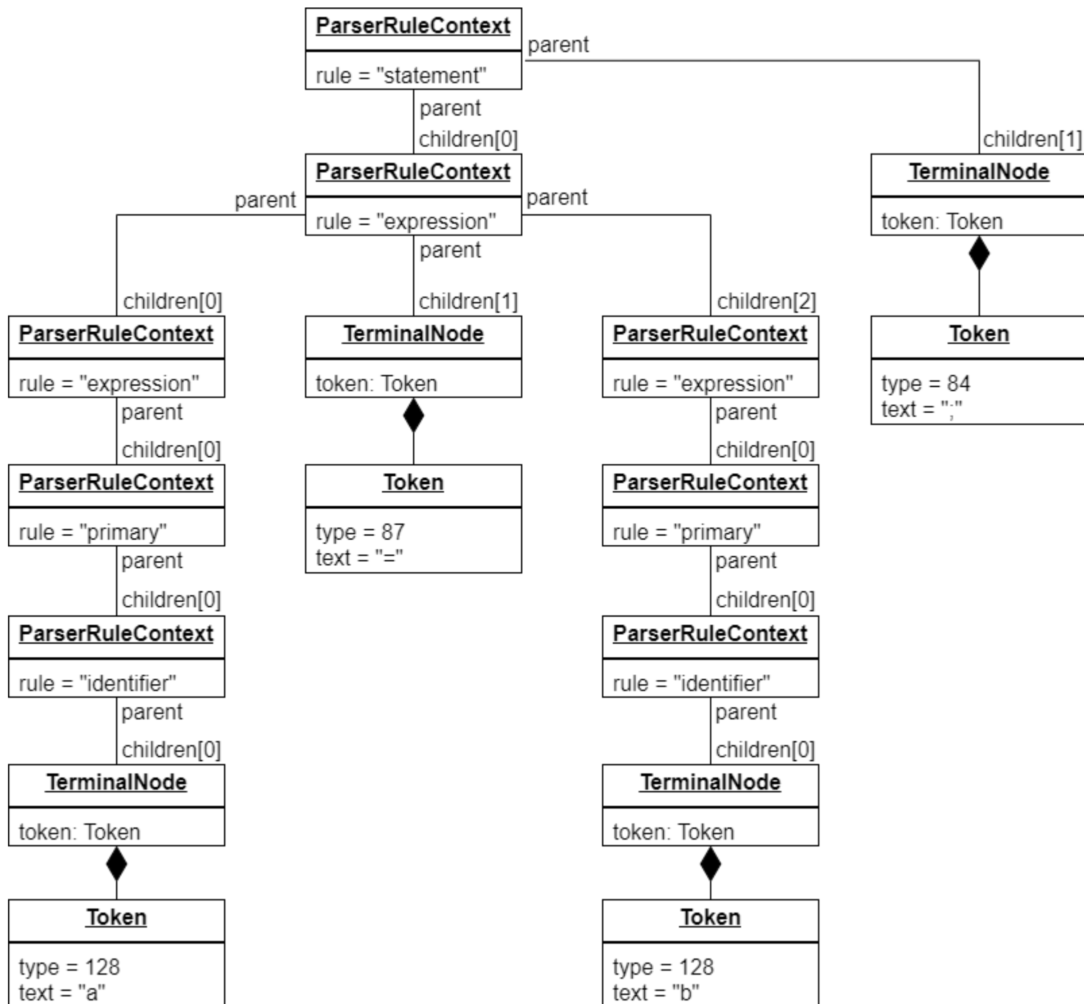

Рис. 15. Синтаксичне дерево, отримане за допомогою синтаксичного аналізатора бібліотеки ANTLR

```
[:statement
 [:expression
  [:expression [:primary [:identifier {:type 128, :text "a"}]]]
  {:type 87, :text "="}
  [:expression [:primary [:identifier {:type 128, :text "b"}]]]]
 {:type 84, :text ";"}]
```

Рис. 16. Синтаксичне дерево з рис. 15 після перетворення функцією try-convert-tree у структуру даних типу «вектор»

Висновки.

В пропонованій статті представлено засіб синтаксичного розбору програмного коду, який базується на використанні популярного генератора синтаксичних аналізаторів ANTLR, а також сучасної функційної мови програмування Clojure, конструкції якої дозволяють виконувати роботу з деревоподібними структурами даних простіше, ніж конструкції нефункційних мов програмування.

На відміну від існуючих бібліотек мови Clojure, розроблений засіб не використовує генератор ANTLR у режимі інтерпретатора, оскільки цей режим є ефективним тільки у обмеженій кількості випадків. Завдяки тому, що програмний код, написаний мовою Clojure, так само як і код, написаний мовою Java, виконується у віртуальній машині JVM (Java Virtual Machine), у розробленому засобі було використано класи бібліотек генератора ANTLR, що написані мовою Java. Це дозволило вирішити задачу реалізації доступу до генератора синтаксичних аналізаторів ANTLR з програмного коду, написаного мовою Clojure.

У якості напрямків подальших досліджень можна запропонувати такі:

1) виконання аналізу методів зіставлення з шаблоном з метою додавання такої можливості до запропонованого засобу;

2) розробка методу конструювання нових дерев синтаксичного розбору на основі вже існуючих дерев.

Список бібліографічного опису

1. Генератор синтаксичних аналізаторів ANTLR. Режим доступу: https://www.antlr.org. Дата звернення: 14.09.2024.
2. Lex & Yacc. Режим доступу: https://epaperpress.com/lexandyacc/. Дата звернення: 14.09.2024.
3. M. Mernik, M. Lenic, E. Avdicausevic, V. Zumer, «Compiler/interpreter generator system LISA», the 33rd Annual Hawaii International Conference on System Sciences, Maui, HI, USA, 2000. DOI: https://doi.org/10.1109/hicss.2000.927021. Дата звернення: 14.09.2024.
4. D. da Cruz, M.J.V. Pereira, M. Béron, R. Fonseca, P.R. Henriques, «Comparing Generators for Language-based Tools», the 1st Conf. on Compiler Related Technologies and Applications, CoRTA'07, Portugal, 2007. Режим доступу: http://surl.li/hnkvgz. Дата звернення: 14.09.2024.
5. F. Ortin, J. Quiroga, O. Rodriguez-Prieto, M. Garcia, «An empirical evaluation of Lex/Yacc and ANTLR parser generation tools», Plos one, 17(3), e0264326, 2022. DOI: https://doi.org/10.1371/journal.pone.0264326. Дата звернення: 14.09.2024.
6. Мова програмування Clojure. Режим доступу: https://clojure.org/. Дата звернення: 14.09.2024.
7. A. Sarimbekov, A. Podzimek, L. Bulej, Y. Zheng, N. Ricci, W. Binder, «Characteristics of Dynamic JVM Languages», the 7th ACM Workshop on Virtual Machines and Intermediate Languages, 2013. DOI: https://doi.org/10.1145/2542142.2542144. Дата звернення: 14.09.2024.
8. K. Kingsbury, «Бібліотека мови Clojure clj-antlr», Github. Режим доступу: https://github.com/aphyr/clj-antlr. Дата звернення: 14.09.2024.
9. M. Engelberg, «Бібліотека мови Clojure instaparse», Github. Режим доступу: https://github.com/Engelberg/instaparse. Дата звернення: 14.09.2024.
10. Antlr Project, «Набір граматик для генератора синтаксичних аналізаторів ANTLR v4», Github. Режим доступу: https://github.com/antlr/grammars-v4. Дата звернення: 14.09.2024.
11. P. Hagelberg, «Засіб Leiningen». Режим доступу: https://leiningen.org/. Дата звернення: 14.09.2024р.
12. G. McCluskey, «Using Java Reflection», Технічна стаття, Oracle, 1998. Режим доступу: http://surl.li/bwgrjz. Дата звернення: 14.09.2024.

References

1. ANTLR parser generator. Access mode: https://www.antlr.org. Accessed on: 14.09.2024.
2. Lex & Yacc. Access mode: https://epaperpress.com/lexandyacc/. Accessed on: 14.09.2024.
3. M. Mernik, M. Lenic, E. Avdicausevic, V. Zumer, «Compiler/interpreter generator system LISA», the 33rd Annual Hawaii International Conference on System Sciences, Maui, HI, USA, 2000. DOI: https://doi.org/10.1109/hicss.2000.927021. Accessed on: 14.09.2024.
4. D. da Cruz, M.J.V. Pereira, M. Béron, R. Fonseca, P.R. Henriques, «Comparing Generators for Language-based Tools», the 1st Conf. on Compiler Related Technologies and Applications, CoRTA'07, Portugal, 2007. Access mode: http://surl.li/hnkvgz. Accessed on: 14.09.2024.
5. F. Ortin, J. Quiroga, O. Rodriguez-Prieto, M. Garcia, «An empirical evaluation of Lex/Yacc and ANTLR parser generation tools», Plos one, 17(3), e0264326, 2022. DOI: https://doi.org/10.1371/journal.pone.0264326. Accessed on: 14.09.2024.
6. The Clojure programming language. Access mode: https://clojure.org/. Accessed on: 14.09.2024.
7. A. Sarimbekov, A. Podzimek, L. Bulej, Y. Zheng, N. Ricci, W. Binder, «Characteristics of Dynamic JVM Languages», the 7th ACM Workshop on Virtual Machines and Intermediate Languages, 2013. DOI: https://doi.org/10.1145/2542142.2542144. Accessed on: 14.09.2024.
8. K. Kingsbury, «The Clojure language library clj-antlr», Github. Access mode: https://github.com/aphyr/clj-antlr. Accessed on: 14.09.2024.
9. M. Engelberg, «The Clojure language library instaparse», Github. Access mode: https://github.com/Engelberg/instaparse. Accessed on: 14.09.2024.
10. Antlr Project, «A set of grammars for the ANTLR v4 parser generator», Github. Access mode: https://github.com/antlr/grammars-v4. Accessed on: 14.09.2024.
11. P. Hagelberg, «The Leiningen tool». Access mode: https://leiningen.org/. Accessed on: 14.09.2024р.
12. G. McCluskey, «Using Java Reflection», Technical Article, Oracle, 1998. Access mode: http://surl.li/bwgrjz. Accessed on: 14.09.2024.

© Касянчук Д.П., Марченко О.І.