Санкт-Петербургский государственный университет

Кафедра системного программирования

Группа 23.Б15-мм

Расширение функциональности генератора синтаксических анализаторов ANTLR на язык программирования OCaml

КОТЕЛЬНИКОВА Ксения Андреевна

Отчёт по учебной практике
в форме «Решение»

Научный руководитель:
Косарев Д.С.

Санкт-Петербург
2025

Оглавление

Введение

В настоящее время существует множество задач, для которых необходимы специфические синтаксические анализаторы. К таким задачам относятся разработка компиляторов и интерпретаторов, обработка пользовательских конфигурационных файлов, создание предметно-ориентированных языков программирования, статический анализ кода и прочие. При этом, самостоятельная разработка синтаксического анализатора требует внушительного количества времени. При таком подходе также увеличивается вероятность пропуска ошибок и недочетов в проекте, и на их предотвращение и исправление следует закладывать еще больше времени. Чтобы решить эту проблему, были созданы инструменты, называемые генераторами синтаксических анализаторов. Их задача состоит в формировании кода анализатора на основе запроса пользователя, оформленного в специальном синтаксисе, а также в предоставлении пользователю легкого в освоении интерфейса. Все тестирование результата при этом происходит внутри инструмента в процессе генерации, абстрагированно от пользователя. Он после прохождения проверок получает сразу пригодный к использованию синтаксический анализатор. Стоит еще отметить, что инструменты для создания анализаторов часто поддерживают при генерации хорошие практики, про которые при ручном написании легко забыть, такие, например, как выведение понятных сообщений об ошибках анализа.

Одним из самых распространенных генераторов является ANTLR (Another Tool for Language Recognition) [1]. На данный момент он официально поддерживает восемь императивных языков программирования, а также имеет множество пользовательских дополнений для интеграции других.

Благодаря легкости применения и обилию поддерживаемых выходных языков, при разработке синтаксического анализатора использование ANTLR часто является требованием компании. При этом, актуальная его версия не поддерживает ни одного полноценно функционального языка. Это делает инструмент менее гибким, а также не позволяет

использовать в работе полезные особенности функциональных языков. Кроме того, интеграция кода на одном языке в проект, написанный на другом, может быть сложным и трудозатратным процессом.

Таким образом была поставлена задача о расширении функциональности ANTLR на какой-либо из функциональных языков. В качестве целевого языка был выбран OCaml благодаря наличию в нем одновременно и элементов объектно-ориентированной парадигмы, которые упростят его интеграцию в код ANTLR, и возможностей функциональной парадигмы, которые могут быть полезными для сгенерированного синтаксического анализатора.

1. Постановка задачи

1. реализовать в инструменте ANTLR модули, позволяющие получить первичные файлы синтаксического анализатора на языке OCaml;

2. реализовать библиотеку runtime поддержки сгенерированного кода;

3. протестировать генерацию кода и библиотеку при помощи unit-тестирования;

4. провести апробацию инструмента на реальной задаче.

2. Обзор

Обзор проводился с целью установления преимущества выбранного инструмента по сравнению с аналогами, а также для изучения технологий, необходимых для реализации расширения ANTLR.

2.1. Обзор используемых технологий

2.1.1. OCaml

Генерация кода на OCaml удобна в первую очередь, потому что позволяет бесшовно соединить полученный анализатор с основным телом проекта. Язык был выбран из соображения его распространенности в сфере синтаксического анализа. Такую популярность OCaml получил благодаря возможности использования как инструментов объектно-ориентированного программирования, так и преимуществ функциональной парадигмы. Среди этих преимуществ, например, мощная система типов, позволяющая быстро определять некорректные объявления в коде, а также возможность реализовывать функции высших порядков, которые делают код более декларативным, коротким и легким для понимания. Со временем, к этим удобствам прибавилась накопившаяся база библиотек, разработанных специально для синтаксического анализа и значительно упрощающих разработку.

2.1.2. ANTLR

- Архитектура

  ANTLR можно разделить на 3 логические части.

  Первая включает в себя обработку исходного файла, в котором находится описание грамматики (для четвертой версии ANTLR это файлы с типом .g4). Результатом его является дерево абстрактного синтаксиса лексического и синтаксического анализа. В своей работе я не затрагиваю эту часть, она является общей для всех выходных языков.

Вторая часть включает в себя генерацию файлов Lexer, Parser и нескольких вспомогательных файлов на основе деревьев абстрактного синтаксиса и специального шаблона из библиотеки StringTemplate. При помощи API, предоставляемого библиотекой, в шаблон подставляются значения из дерева абстрактного синтаксиса, что на выходе формирует полноценный файл на целевом языке.

Однако в таком виде синтаксический анализатор еще не готов к работе. Для обработки обработки входной строки ему необходима третья часть инструмента: вспомогательная функциональность, которая заранее должна быть реализована внутри ANTLR в модуле runtime. Особенность модуля заключается в том, что он должен быть реализован полностью на целевом языке, и содержать все необходимые для компиляции конфигурационные файлы. Фактически, runtime должен представлять собой полноценную библиотеку вспомогательных для синтаксического анализатора методов.

- Библиотека StringTemplate [2]

  Генерация файлов синтаксического и лексического анализаторов происходит посредством библиотеки StringTemplate. С помощью нее можно создавать так называемые строковые шаблоны — текст со специально обозначенными местами, в которые нужно будет вставить некоторое поступившее значение. Аргументы, которые будут подставляться в шаблон, называются аттрибутами.

  Библиотека StringTemplate предоставляет исчерпывающий API для взаимодействия с шаблонами, но в контексте ANTLR важнее всего метод add, позволяющий по шаблону, от которого он вызывается, сгенерировать готовую строку с аргументами метода, подставленными вместо аттрибутов.

  Еще одной важной функциональностью в StringTemplate является создание групп шаблонов. Это файлы с типом .stg, внутри которых можно объявить несколько взаимосвязанных шаблонов. Внут-

ри одной группы шаблоны могут подставляться друг в друга, передавая свои значения аттрибутов в качестве аргументов внутреннему шаблону. Именно так и происходит создание синтаксического анализатора: генерация основного класса запускает множество вложенных генераций.

Таким образом, для добавления нового целевого языка необходимо описать группу шаблонов, на основе которых будет создаваться результат.

- Сравнение с другими аналогичными инструментами

  – В отличие от YACC и Bison, генерирующих LALR (lookahead left-to-right) анализаторы, ANTLR поддерживает LL(*) (leftmost derivation с разбором слева направо и заглядыванием вперед на произвольное количество токенов) [4]. Для достижения произвольности количества токенов ANTLR использует конечные автоматы вместо таблиц переходов. В отличие от LR подхода, LL не поддерживает обработку грамматик с леворекурсивными правилами. Чтобы решить эту проблему, ANTLR сканирует введенную в него грамматику на наличие левой рекурсии, и, при обнаружении, переписывает эти правила в эквивалентные с использованием префиксов. Такая система неидеальна: слишком сложные правила ANTLR не способен преобразовать автоматически. В этом случае он уведомит пользователя о необходимости переписать правило. LL-анализ, однако, имеет серьезное преимущество перед LR в обработке ошибок и удобстве использования.

  – ANTLR — самый популярный инструмент для генерации синтаксических анализаторов, и за время его существования накопилось много материалов, помогающих разобраться в его работе. Это выгодно выделяет его на фоне YACC, Bison, PEG.js и других менее популярных решений.

2.1.3. Dune

Модуль runtime должен быть реализован на целевом языке, что, в случае OCaml, ставит вопрос о выборе системы сборки. Dune из всех возможных вариантов является самой легкой в использовании, наряду с OCamlBuild, Jbuilder и OASIS. Кроме того, на данный момент, Dune является стандартом среди OCaml-проектов.

2.2. Обзор существующих решений

2.2.1. Реализованные языки ANTLR

Текущая версия ANTLR[1]. поддерживает генерацию парсеров в 10 языков программирования: C++, C#, Dart, Java, JavaScript, PHP, Python версии 3, Swift, TypeScript и Go. Несмотря на наличие черт функциональной парадигмы в некоторых из них, все эти языки далеки по синтаксису и особенностям реализации от OCaml, что делает затруднительным простой перевод готового анализатора с одного из уже реализованных языков на выбранный целевой. Кроме того, для работы сгенерированный при помощи ANTLR анализатор требует специально реализованный на выходном языке модуль runtime.

Эти обстоятельства делают нецелесообразной идею использования ANTLR для OCaml-проекта без расширения текущей функциональности инструмента.

2.3. Выводы

На данный момент ANTLR является самым широко используемым генератором синтаксических анализаторов, в том числе, по причине удобства расширения. Библиотека StringTemplate помогает полностью освободить разработчика нового выходного языка от необходимости реализовывать все модули, связанные с первичной обработкой файла грамматики. Работа, таким образом, сводится к правильной настройке

[1]Версия 4.13.2 на момент обращения 4 декабря 2024 г.

строкового шаблона и реализации модуля runtime [5], необходимого для работы готового анализатора.

3. Описание решения

3.1. Подготовка файла группы шаблонов

Первая существенная часть работы заключается в реализации файла группы шаблонов. В инструкции по добавлению новых целевых языков от авторов ANTLR присутствует рекомендация о том, что за основу нового файла следует взять один из уже реализованных. В случае OCaml это не настолько выгодно, так как синтаксис слишком отличается от всех представленных в ANTLR языков, но все еще полезно для отслеживания зависимостей шаблонов и общего понимания их смысла.

Работа с группой шаблонов серьезно замедляется из-за отсутствия инструментов форматирования и статического анализа файлов типа .stg. Ориентироваться в коде затруднительно, а возникающие ошибки выявляются только после тестового запуска, на который уходит около полуминуты на сборку проекта.

Кроме того, сообщения об ошибках, которые выдает ANTLR, не всегда информативны. В библиотеке StringTemplate есть встроенная система обработки ошибок, однако иногда ее сообщения давали неправильное представление о возникшей ошибке. Например, ошибка, описанная как отсутствие определенного шаблона, на самом деле произошла из-за лишнего управляющего символа в совершенно другом шаблоне.

Также в ANTLR при взаимодействии с файлом группы шаблонов никак не обрабатывается исключение разыменования нулевого указателя. Это затрудняет локализацию ошибки, так как без дополнительных инструментов невозможно понять, какой именно объект не смог создаться и вызвал исключение.

В процессе работы пришлось изучить документацию StringTemplate из-за необходимости использовать внутри шаблона символ «<», являющийся в этой библиотеке управляющим.

Готовый файл группы шаблонов был помещен в директорию tool/templates/codegen/OCaml, как и было описано в инструкции по добавлению новых целевых языков.

3.2. Реализация модуля runtime

В ANTLR модуль runtime представляет собой обширную библиотеку, включающую в себя следующие большие логические блоки: работа с ATN (augmented transition networks) [3] и детерминированными конечными автоматами, построение на их основе дерева абстрактного синтаксиса, работа с потоком токенов и классы Lexer и Parser, в которых реализована функциональность, общая для всех сгенерированных на основе грамматики модулей.

Основная проблема на этом этапе работы состояла в сложности погружения в код. Runtime практически не имеет файловой структуры, а та, которая есть, не всегда логически интуитивна. Кроме того, на код в большинстве отсутствует документация. Ни названия классов, ни какие-либо другие признаки не помогают разбраться в зависимости частей кода друг от друга, а отсутствие какого-либо разделения на блоки приводит к высокой сложности определения взаимосвязи между классами. Говоря о названиях, они зачастую еще больше уменьшали читаемость.

В ходе решения задачи была реализована часть модуля runtime, необходимая для поддержки работы простых числовых грамматик. В дальнейшем возможно расширение библиотеки до полной функциональности, представленной в частях ANTLR для других языков.

3.3. Добавление нового файла target [6]

Для того, чтобы сконфигурировать ANTLR для работы с новым целевым языком, необходимо создать файл с классом OCamlTarget, в котором указать ключевые слова языка и управляющие последовательности. На первую часть имени этого класса ANTLR будет ориентироваться при поиске файла строкового шаблона, поэтому шаблон необходимо было назвать соответственно: OCaml.stg.

4. Тестирование

4.1. Юнит-тестирование runtime

Основные функции модуля runtime были протестированы внутри OCaml-проекта при помощи unit-тестирования.

4.2. Тестирование строкового шаблона

Работа строкового шаблона была протестирована unit-тестами, а также на простой грамматике, для которой достаточно реализованной части runtime.

4.3. Непрерывная интеграция

В ходе разработки к написанному на OCaml коду применялся инструмент форматирования ocamlformat и инструмент статического анализа zanuda.

Заключение

- Реализовано порождение кода синтаксического анализатора на языке OCaml по файлу грамматики;
- Реализовано подмножество библиотеки runtime для поддержки сгенерированного кода;
- Полученные результаты протестированы unit-тестами, ocamlformat и zanuda;
- Простая задача синтаксического анализа решена.

В дальнейшем могут быть реализованы следующие задачи:

- Расширение возможностей библиотеки runtime
- Автоматизация интеграционного тестирования инструмента

Исходный код работы доступен на Github [7].

Список литературы

[1] Parr T. ANTLR. — URL: https://github.com/antlr/antlr4/blob/master/doc/index.md. Проверено: 2025-01-01.

[2] Parr T. Библиотека StringTemplate. — URL: https://github.com/antlr/stringtemplate4/blob/master/doc/index.md. Проверено: 2025-01-01.

[3] T. Parr S.Harwell K.Fisher. Adaptive LL(*) Parsing: The Power of Dynamic Analysis. — 2014.

[4] T. Parr R.Quong. ANTLR: A predicated-LL(k) parser generator // Software: practice and experience. — 1995. — Vol. 25.

[5] runtime API. — URL: https://javadoc.io/doc/org.antlr/antlr4-runtime/latest/index.html. Проверено: 2025-01-01.

[6] target API. — URL: https://www.antlr.org/api/JavaTool/org/antlr/v4/codegen/Target.html. Проверено: 2025-01-01.

[7] Репозиторий с кодом. — URL: https://github.com/p1onerka/antlr4-OCamlTarget/tree/dev.