Санкт-Петербургский государственный университет

Асеев Григорий Александрович

Выпускная квалификационная работа

Ускорение фильтрации трафика антибот-системы с использованием библиотеки исполнения выражений

Уровень образования: бакалавриат

Направление *09.03.04 «Программная инженерия»*

Основная образовательная программа *СВ.5080.2021 «Программная инженерия»*

Научный руководитель:
доцент кафедры системного программирования, к.ф.-м.н., Луцив Д. В.

Рецензент:
Программист ООО «Озон Технологии» Гришанков Е. М.

Санкт-Петербург
2025

Saint Petersburg State University

Aseev Grigory

Bachelor's Thesis

Acceleration of traffic filtering of antibot-system using expression execution library

Education level: bachelor

Speciality *09.03.04 «Software Engineering»*

Programme *CB.5080.2021 «Software Engineering»*

Scientific supervisor:
C.Sc., docent D. V. Luciv

Reviewer:
Programmer at «Ozon Technologies» E. M. Grishankov

Saint Petersburg
2025

Оглавление

Введение

Фильтрация интернет-трафика является одной из ключевых задач современных антибот-систем, особенно в условиях постоянно растущего числа автоматизированных угроз, таких как *DDoS*[2]-атаки. Для обеспечения надежной защиты важно минимизировать нагрузку на вычислительные ресурсы системы, обеспечивая при этом оперативную и точную фильтрацию трафика.

В компании *Ozon*[7] для фильтрации трафика используется антибот-система, основанная на агентах — микросервисах, написанных на *Golang*[4]. Одним из ключевых компонентов системы является *ручной рейт-лимитер*[10] (сенсор), который позволяет создавать и применять правила в виде логических выражений на основе параметров запросов. Эти правила используются для защиты от нежелательного трафика и могут приводить к санкциям, включая блокировки. Аналитики данных участвуют в разработке правил и вносят в них в среднем 5 изменений в день. С августа 2023 года по апрель 2025 года количество таких правил увеличилось с 6 до 529, что существенно повысило нагрузку на агентов. В связи с этим возникает необходимость в дальнейшей оптимизации исполнения правил для обеспечения масштабируемости системы без перегрузки вычислительных ресурсов.

Гипотеза исследования заключается в том, что применение компиляции правил в функции-предикаты с использованием замыканий, передача переменных через массив значений во время исполнения и вынесение проверок типов на этап компиляции позволит значительно ускорить обработку правил. В текущей реализации ручного рейт-лимитера используется библиотека *Expr*[3] на языке Golang, которая применяет байткод-компиляцию и стековую модель выполнения инструкций. Несмотря на то что Expr является одной из самых быстрых библиотек для подобных задач, предлагаемый подход с замыканиями потенциально может обеспечить более высокую производительность при работе с большим количеством сложных правил. Это достигается за счет ленивых вычислений и оптимизации доступа к данным через функциональ-

ный подход. Проверка данной гипотезы требует экспериментального сравнения производительности двух методов на реальных сценариях работы антибот-системы Ozon.

Целью данной работы является разработка высокопроизводительной библиотеки для выполнения правил с синтаксисом, совместимым с существующей системой антибота Ozon. В рамках исследования планируется провести сравнительный анализ предложенного решения с открытыми аналогами, оценив производительность и потребление памяти. Ключевым этапом работы станет внедрение разработанной библиотеки в сервисы антибот-системы для проверки её эффективности в реальных условиях. Успешная реализация проекта позволит снизить вычислительную нагрузку и повысить масштабируемость системы при дальнейшем росте количества правил.

Дата сборки: 28 мая 2025 г.

1. Постановка задачи

Целью данной работы является разработка библиотеки для высокопроизводительной фильтрации трафика в антибот-системах, обеспечивающей совместимость с текущими правилами антибота Ozon. Для достижения цели были поставлены следующие задачи:

1. Описать процесс фильтрации трафика с использованием правил в антибот-системе.
2. Провести анализ существующих библиотек и подходов для исполнения выражений.
3. Спроектировать библиотеку высокопроизводительной фильтрации трафика в антибот-системах.
4. Выполнить реализацию указанной библиотеки, включающую:
   - Парсер выражений;
   - Компилятор преобразующий выражения в замыкания;
   - Механизм доступа к переменным;
   - Обеспечение полной совместимости синтаксиса с текущими правилами антибот-системы;
5. Разработать набор тестов для библиотеки, интегрировать её в сервисы и провести сравнительный анализ производительности с существующими решениями.

2. Обзор

Обзор включает в себя процесс фильтрации в разрабатываемой антибот-системе, описание библиотек для исполнения правил, их недостатков, а также общую концепцию эвалюатора, основанного на замыканиях.

2.1. Фильтрация трафика с использованием правил

В антибот-системе Ozon, реализованной в микросервисной архитектуре на языке Golang, фильтрация трафика осуществляется на основе динамически обновляемых правил. Аналитики данных взаимодействуют с панелью управления, через которую создают и корректируют правила, внося в среднем по 5 изменений в день. Изменения сохраняются в централизованном хранилище при помощи сервиса RulesUpdater. Агент антибота периодически загружает актуальные правила из хранилища и компилирует их с использованием специализированной библиотеки исполнения. При поступлении HTTP-запроса агент извлекает из него ключевые данные и применяет к ним скомпилированные правила. На основе полученных результатов система принимает решение о блокировке подозрительного трафика или пропуске запроса.

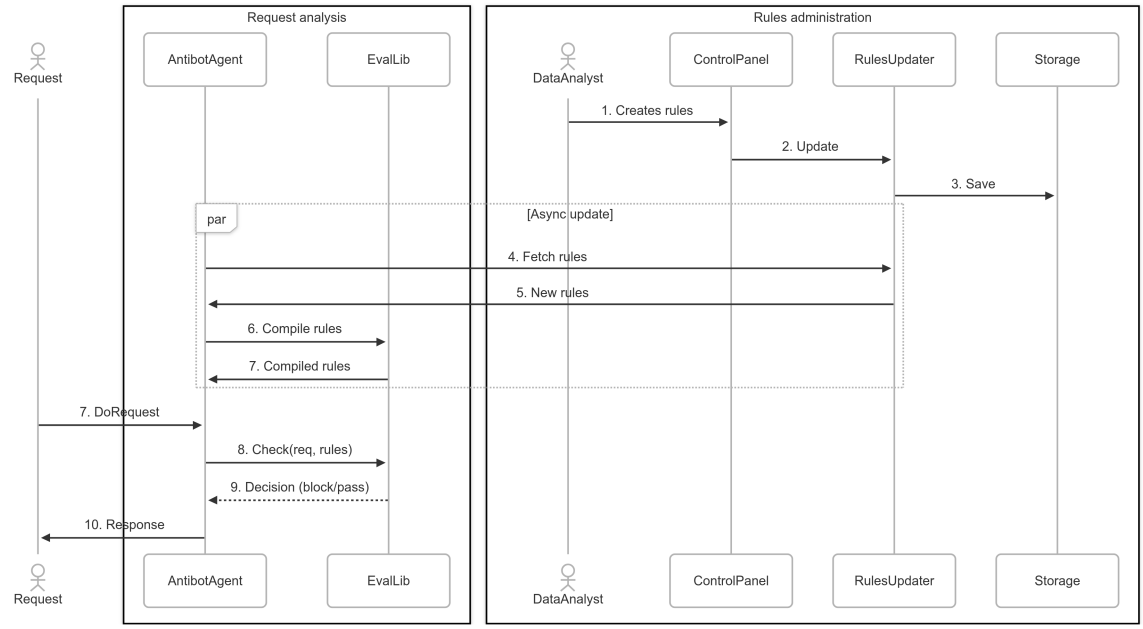

Рис. 1: Диаграмма последовательностей фильтрации трафика

2.2. Существующие библиотеки и подходы исполнения выражений

2.2.1. Govaluate

Govaluate[6] — это библиотека для языка программирования Go, предназначенная для парсинга и вычисления выражений во время выполнения. Она представляет собой интерпретатор выражений, который позволяет работать с математическими и логическими операциями без необходимости компиляции кода. Govaluate широко используется в системах, где требуется обработка динамических выражений, таких как конфигурируемые бизнес-правила, вычисления на лету и пользовательские фильтры.

Одной из ключевых возможностей Govaluate является работа с переменными. В выражениях можно использовать значения, передавая их в виде хеш-таблицы с значениями общего типа. Это позволяет подставлять данные в выражение во время выполнения. Кроме того, библиотека поддерживает работу с полями структуры, передаваемой в качестве переменной.

Библиотека поддерживает широкий набор математических операторов. Также доступен тернарный оператор, что дает возможность создавать более сложные вычисления и условные конструкции в выражениях. Дополнительно Govaluate поддерживает добавление пользовательских функций. Можно зарегистрировать свою функцию и использовать ее в выражениях, что расширяет возможности библиотеки и позволяет адаптировать ее под конкретные задачи. Это особенно полезно в случаях, когда стандартных операторов недостаточно, и требуется выполнение специфической логики.

Govalute является интерпретатором выражений, что накладывает ограничения на его производительность. Он каждый раз заново вычисляет выражение, проходя AST, вместо того чтобы заранее оптимизировать вычисления. Это делает его неэффективным в сценариях многократного вычисления одного и того же выражения и накладывает дополнительные издержки.

2.2.2. Expr

Expr[3] — это высокопроизводительный эвалюатор выражений для Go, который выгодно отличается от интерпретаторов, таких как Govalute, благодаря своей архитектуре. В его основе лежат компиляция в байткод и оптимизации на этапе анализа выражения, что позволяет значительно повысить скорость выполнения.

На этапе разбора выражений Expr применяет различные оптимизации: константное свёртывание — выражения, такие как 1 + 2, вычисляются заранее, вызов функций с заранее известными аргументами. Это снижает нагрузку во время исполнения.

Expr поддерживает математические и логические операторы, встроенные функции для работы со строками, числами и булевыми выражениями, а также операции с массивами и структурами. Кроме того, он позволяет добавлять пользовательские функции, расширяя возможности выражений. Expr показывает высокую производительность за счёт оптимизирующей компиляции и исполнения байткода на виртуальной машине, которую можно переиспользовать, что даст прирост к скорости исполнения на 10-15%. Это подтверждается результатами бенчмарков[1]. Благодаря своей скорости и гибкости, он широко используется в различных компаниях для обработки выражений в таких, как *Google*[5] и *Uber*[11].

При использовании данного подхода требуется предварительное вычисление значений всех переменных перед их передачей в эвалюатор, вне зависимости от того, используются ли они в выражении. Это может привести к избыточным вычислениям (таким, как парсинг User-Agent и внутренних токенов) большого числа (порядка нескольких сотен) сложных переменных. Дополнительно, Expr работает с типом any (обобщенный тип данных в языке Golang), это влечёт за собой дополнительные приведения типов во время выполнения. В языке Go строки являются неизменяемыми, поэтому их приведение к any приводит к копированию строки, что увеличивает время выполнения и вызывает дополнитель-

[1] https://github.com/antonmedv/golang-expression-evaluation-comparison#readme

ные аллокации в памяти. В совокупности эти факторы могут негативно сказаться на производительности при частом вычислении выражений с динамическими данными.

2.2.3. Подход на основе замыканий

Компиляция правил в функции-предикаты с использованием замыканий, хранение переменных в массиве вместо хеш-таблицы и вынесение проверок типов на этап компиляции позволяют существенно повысить производительность вычислений по сравнению с эвалюатором, использующим байткодовое исполнение. Такой подход устраняет накладные расходы на интерпретацию, снижает затраты на доступ к переменным за счёт использования индексов вместо строковых ключей и исключает необходимость приведения типов во время выполнения.

Дополнительно, в традиционных эвалюаторах строки при передаче приводятся к общему типу, что приводит к их копированию и, соответственно, к дополнительным аллокациям, особенно при обработке длинных значений, таких как User-Agent и фингерпринт, в антибот-системах на Golang. В типизированном эвалюаторе, построенном на основе замыканий, строки передаются без лишнего копирования, что снижает нагрузку на память и ускоряет выполнение выражений.

В статье *Building Fast Interpreters in Rust*[1] рассматриваются преимущества использования эвалюатора на основе замыканий. Замыкание (Fn box)[2] инкапсулирует все необходимые данные окружения, требуемые для вычисления выражения, включая вспомогательные функции (fn), как пользовательские функции, операторы, а также значения переменных, полученные во время выполнения (RuntimeHashValues), и вложенные замыкания (other Fn), из которых оно состоит.

[2] https://blog.cloudflare.com/building-fast-interpreters-in-rust/

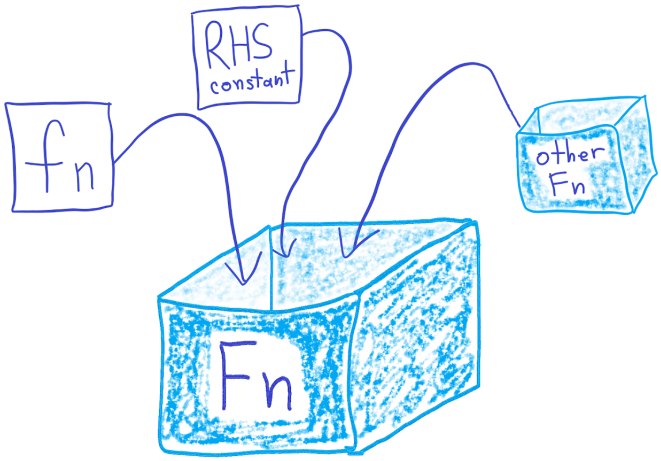

Рис. 2: Выражение в виде замыканий

Компиляция выражений в замыкания позволяет значительно ускорить обработку часто встречающихся выражений за счёт снижения накладных расходов на интерпретацию. Несмотря на создание множества небольших объектов, их влияние на производительность остаётся незначительным, а отказ от генерации кода в runtime способствует повышению безопасности и предсказуемости работы. В результате данный метод обеспечивает прирост производительности на 10–15% по сравнению с интерпретацией AST.

В статье представлены сравнительные измерения производительности различных подходов[3], включая интерпретацию AST, компиляцию в байткод и компиляцию в замыкания. Анализ результатов в зависимости от времени выполнения (в наносекундах) и количества запусков выражений показывает, что использование замыканий демонстрирует наилучшую скорость вычислений.

[3] https://www.slideshare.net/slideshow/building-fast-interpreters-in-rust/125271384

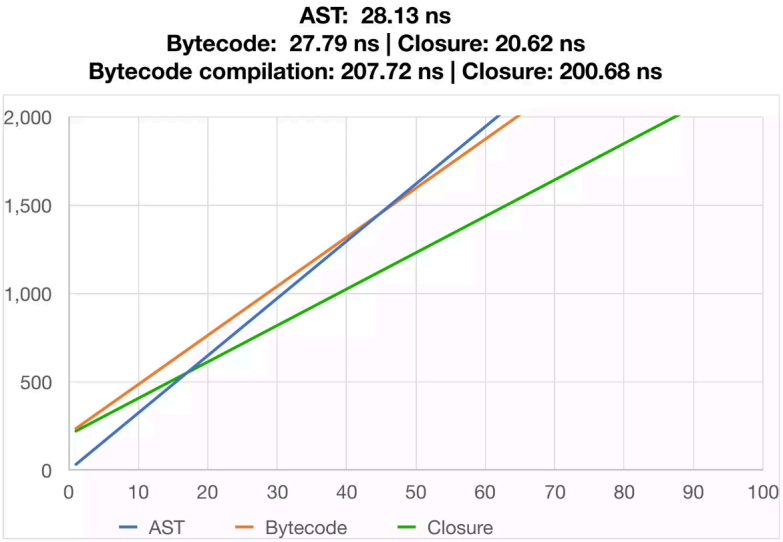

Рис. 3: Сравнение исполнения выражений

3. Архитектура

В данном разделе представлена архитектура компонентов библиотеки и их функциональное назначение.

3.1. Компоненты библиотеки исполнения правил

Для разбиения строки на токены реализован Lexer, который хранит строку в виде буффера символов и определяет последовательно за O(N) их лексическую семантику. Токены подразделяются на два типа: базовые — без дополнительной информации (операторы, знаки препинания) и составные — хранят в себе значение токена (литералы, идентификаторы). На изображении 4 представлена диаграмма классов данного компонента.

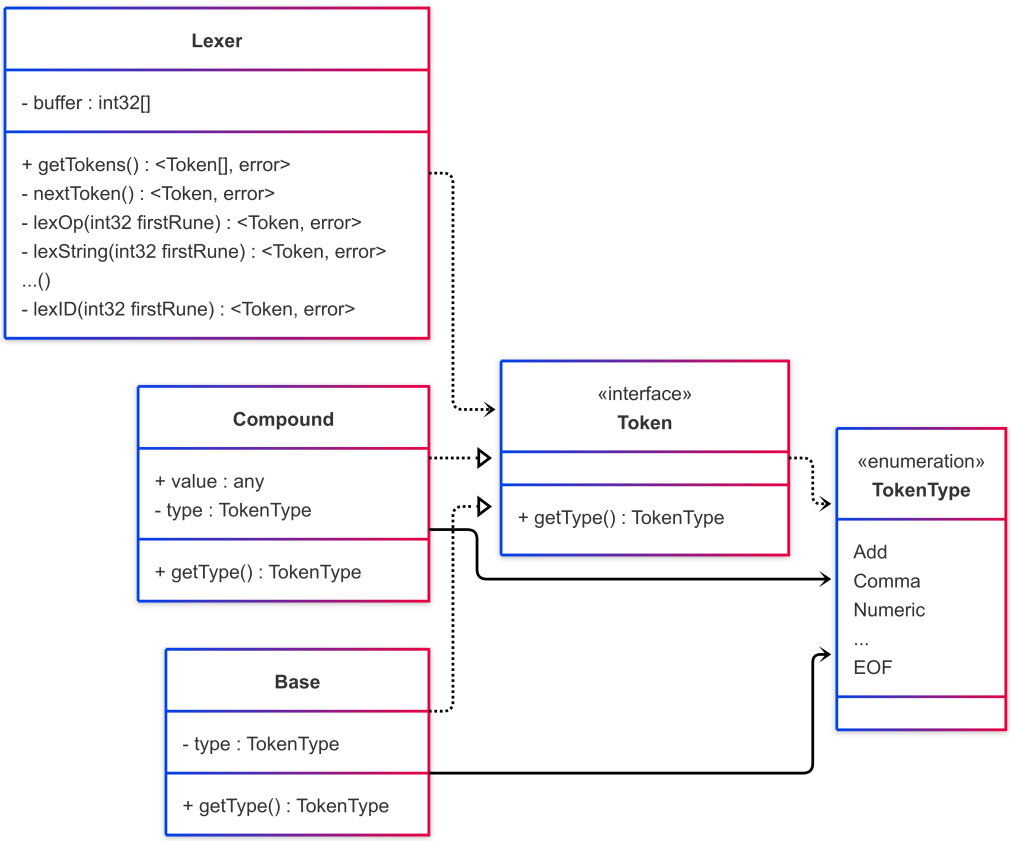

Рис. 4: Диаграмма классов лексера

Чтобы построить AST выражения был спроектирован PrattParser. Узел представляет собой интерфейс, который принимает объект реализующий паттерн Visitor и позволяет посетить себя данному Visitor, вызвав соотвествующий метод. PrattParser, используя Tokenizer, разбивает строку на токены и с помощью таблицы их приоритетности строит искомые узлы рекурсивно. Компонент парсера изображен на рис. 5.

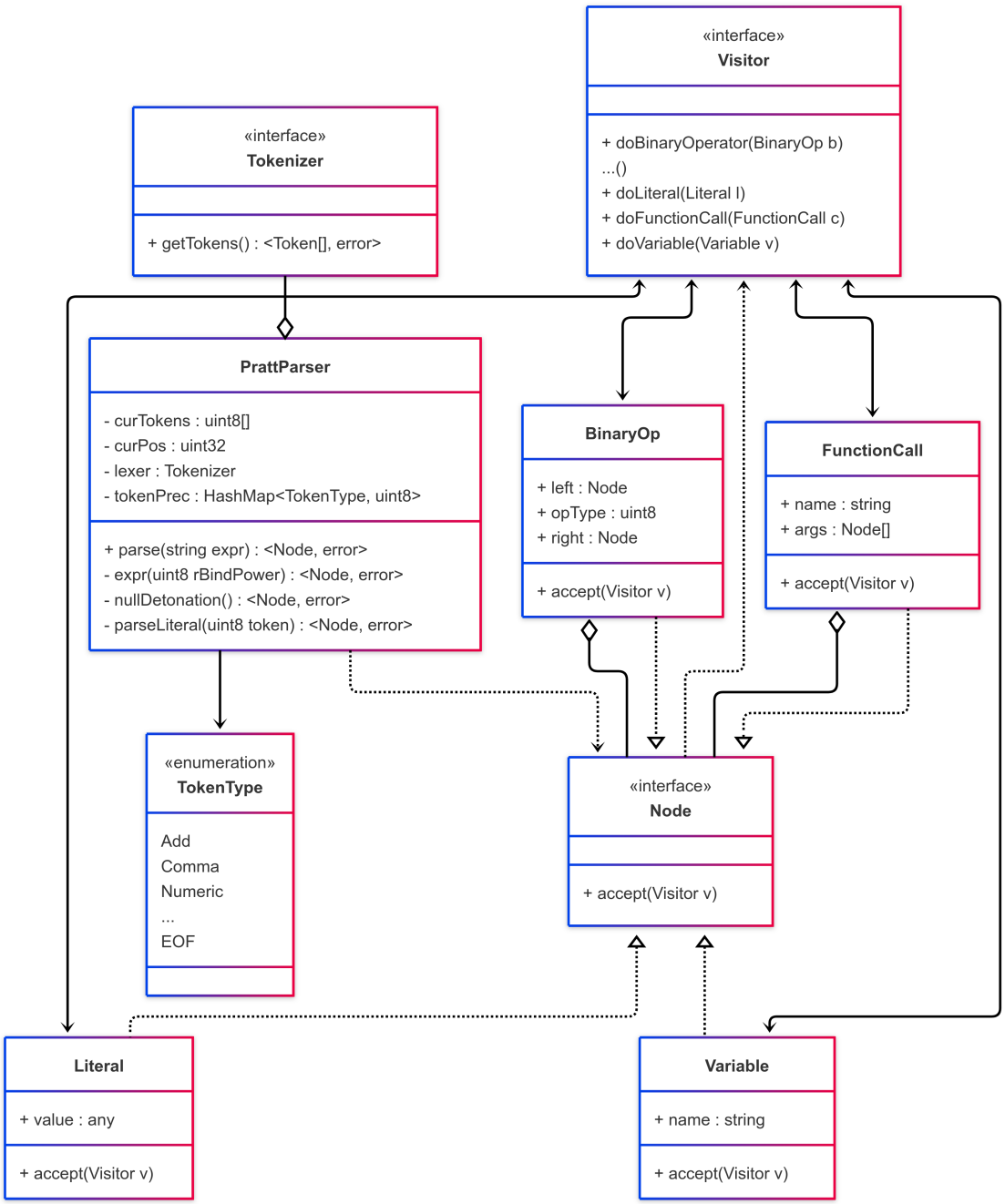

Рис. 5: Диаграмма классов парсера

Для построения замыкания Builder реализует паттерн Visitor, который хранит информацию о типе и значении переменных и пользовательских функций. Результирующее замыкание представлено объектом Value, содержащим скомпилированные функции, ожидающие провайдера переменных для выполнения. Пользователь может создавать функции, в аргументы которых попадают значения в виде замыканий, а

тип функции может иметь множество реализуемых сигнатур с variadic аргументом. На изображении 6 представлена диаграмма классов компиляции выражения.

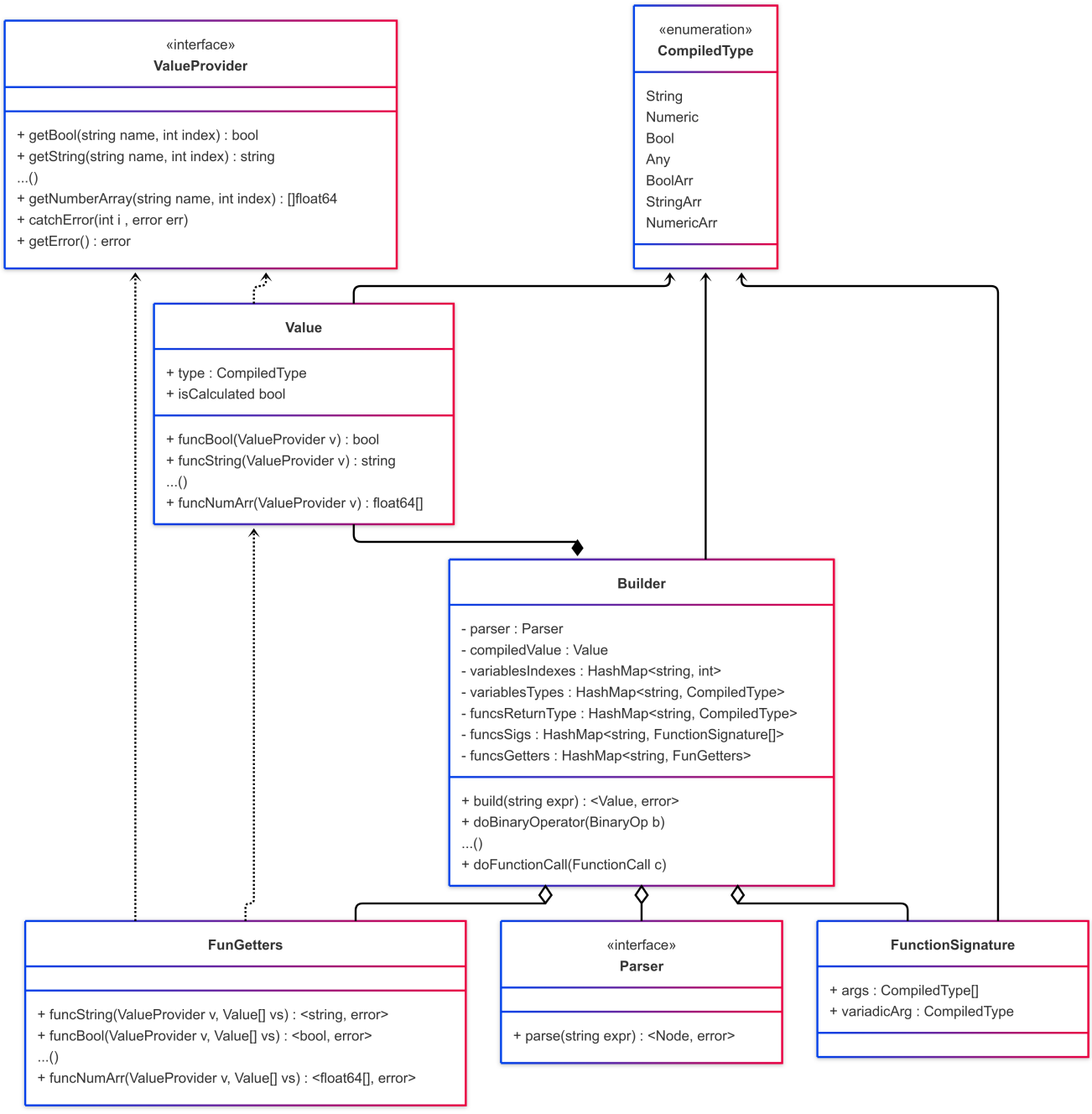

Рис. 6: Диаграмма классов компилятора

Провайдером переменных служит интерфейс ValueProvider, который предоставляет значения переменных в момент его исполнения. MapValueProvider реализует данный интерфейс, храня значения в виде хеш-таблицы с обобщенным типами. Она предоставляет удобный для пользователя процесс передачи переменных. При этом другая его реализация, FuncSliceValueProvider, хранит специальные Getter-функции, которые принимают пользовательский аргумент для вычисления значений переменных. Пользователю требуется хранить индексы переменных и создавать типизированные реализации переменных, но это при этом скорость исполнения будет выше за счет этого. Провайдер переменных представлен на диаграмме классов (рис. 7).

Рис. 7: Диаграмма классов провайдера переменных

4. Особенности реализации

В данном разделе представлена реализация эвалюатора на основе замыканий и описание поддерживаемого языка выражений. Его структура включает несколько уровней, на каждом из которых выполняются этапы преобразования выражения, завершающиеся компиляцией в замыкание.

4.1. Алгоритм парсера выражений

Эвалюатор выражений включает в себя токенайзер и парсер, выполняющие разбиение входной строки и её синтаксический анализ. Токенайзер преобразует выражение в последовательность токенов, определяя знаки препинания, литералы, переменные и операторы. Парсер, в свою очередь, анализирует полученные токены и формирует из них узлы абстрактного синтаксического дерева (AST), обеспечивая структурированное представление выражения для дальнейшей обработки.

По требованиям необходимо было разработать собственный парсер, чтобы библиотека не зависела от сторонних импортов и чужих решений, что обеспечивает полный контроль над кодом, избегает лицензионных конфликтов, минимизирует риски уязвимостей и упрощает поддержку.

Для синтаксического анализа выбран алгоритм *Pratt Parser*[9], который отличается гибкостью и эффективностью при работе с грамматическими конструкциями, особенно в контексте математических операций и вызовов функций. Он позволяет корректно обрабатывать операции с различными приоритетами и ассоциативностью, а также эффективно управлять скобками и операторами. Благодаря своей архитектуре алгоритм легко адаптируется для поддержки новых типов данных и операций. Также упрощает обработку ошибок и делает более детальным их вывод при построении узлов. Это делает его оптимальным выбором для рассматриваемого эвалюатора.

Основные механизмы Pratt Parser:

- Null Denotation (nud) — создаёт узлы начального контекста, включая литералы и унарные операции.
- Left Denotation (led) — формирует узлы с инфиксным контекстом, такие как бинарные и тернарные операторы.
- Left Binding Power (lbp) — определяет приоритет связующего токена, регулируя порядок вычислений.

Pratt Parser использует механизм рекурсивного нисходящего разбора с управляемым порядком обработки операторов. Вместо традиционных грамматических правил он применяет функции обработки токенов, основанные на их приоритете.

4.2. Построение замыкания

Компонент Builder отвечает за компиляцию абстрактного синтаксического дерева в замыкание — функцию, обладающую окружением с заранее вычисленными значениями. Скомпилированное выражение готово к выполнению и требует только провайдера переменных для окончательного вычисления. Такой подход позволяет исключить накладные расходы на интерпретацию AST в момент исполнения, повышая производительность за счёт предвычисления.

Каждый узел AST возвращает своё значение в виде замыкания явного типа: func(P) Type, где P — интерфейс, предоставляющий доступ к окружению переменных во время выполнения, а Type — один из предопределённых типов (bool, string, float64, []bool, []string, []float64 или any). Последний вариант используется исключительно для передачи кастомных структур через окружение или переменные в пользовательские функции. Такой функциональный подход обеспечивает эффективное исполнение выражений и предотвращает избыточное копирование строк.

Оптимизации заключаются в жадной компиляции узлов и предварительной проверке типов вне замыкания. Это позволяет вынести все возможные вычисления из этапа выполнения, что ускоряет обработку

выражений в момент исполнения. При компиляции пользователь предоставляет информацию о типах переменных и функций, а также индекс переменной при использовании провайдера на основе слайса. Владение этой информацией позволяет не только безопасно проверять типы, но и корректно строить замыкания, поскольку тип результата необходимо определить заранее.

Например, условие:

Domain in [”globaloet”+”.ru”, ”ozon.ru”] && rate(”1m”, IP) $>$ 1000

Компилируется в:

And(In(Domain(P),Arr(”globaloet.ru”, ”ozon.ru”)), Gt(Rate(”1m”, IP(P)), 1000))

Для выполнения скомпилированного выражения требуется предоставить ValueProvider, который динамически подставит значения переменных в процессе вычисления. Такой метод обеспечивает высокую производительность и предсказуемость работы эвалюатора.

В листинге 1 можно увидеть пример компиляции массива в замыкание:

Листинг 1: Функция преобразует массив ast в замыкание

```
func processArray[V string | float64 | bool](
        constGetter,
        nonConstGetter map[int]func(varprov.ValueProvider) V,
        ) (*Value, error) {
        res := make([]V, len(constGetter)+len(nonConstGetter))
        for i, v := range constGetter {
                res[i] = v(nil)
        }
        if len(nonConstGetter) == 0 {
                return wrapResultValue(
                        true,
                        func(_ varprov.ValueProvider) []V {
                        return res
                })
        }

        return wrapResultValue(
                false,
                func(r varprov.ValueProvider) []V {
                for i, v := range nonConstGetter {
                        res[i] = v(r)
                }
                return res
        })
}
```

Также в листинге 2 можно увидеть, как задаются свои кастомные функции. Пользователь указывает явные типы сигнатуры функции и её реализацию, в аргументы которой будут попадать скомпилированные замыкания. Это даёт возможность гибко расширять функциональность и создавать выражения, которые могут быть вычислены с использованием пользовательских логик и вычислений.

Листинг 2: Пользовательская функция contains

```
func containsFeev() (evaluator.FunctionProvider, error) {
        return evaluator.NewFunctionProvider(
                evaluator.NewCalleeInfo(
                        evaluator.NewFunctionSignature(
                                evaluator.NewValue(
                                        evaluator.String,
                                        false,
                                ),
                                evaluator.NewValue(
                                        evaluator.String,
                                        false,
                                ),
                        ),
                ),
    func(r feev.ValueProvider, args ...*feev.Value) (bool, error) {
            return strings.Contains(
                args[0].Base.String(r),
                args[1].Base.String(r),
            ), nil
    },
        )
}
```

4.3. Доступ к переменным

ValueProvider представляет собой компонент, который хранит значения всех переменных и предоставляет доступ к этим значениям во время выполнения скомпилированного замыкания. Он использует хеш-таблицу, чтобы извлечь значения переменных по их имени. Однако в худшем случае время доступа к данным может составлять O(N), что ограничивает эффективность при работе с большим количеством переменных.

Для улучшения производительности предлагается использовать альтернативный подход с массивом, в котором каждому значению переменной присваивается индекс. Индексы задаются в процессе компиляции в Builder, и значения передаются в установленном порядке. Такой метод

обеспечивает более быструю асимптотику доступа — O(1). Кроме того, для оптимизации значения хранятся в виде типизированных функций, что позволяет избежать вычислений для тяжелых переменных, не используемых в выражении, повышая тем самым общую эффективность выполнения.

4.4. Грамматика языка

В связи со спецификой языка антибот-системы поддерживаются базовые типы данных, такие как float64, bool, string, а также массивы, содержащие эти базовые типы. Для передачи кастомных структур предусмотрен тип any. В правилах антибот-системы Ozon использовалось подмножество DSL библиотеки Expr, поэтому разрабатываемая библиотека проектировалась, чтобы ее грамматика содержала данное подмножество. Описываемая грамматика языка с поддерживаемыми типами и операторами представлена на изображении 8:

```
expr = ternary;

ternary = log_or
| log_or "?" expr ":" expr;

log_or = log_and
| log_or "||" log_and;

log_and = in
| log_and "&&" in;

in = bit_or
| in ("in" | "not in") bit_or;

bit_or = xor
| bit_or "|" xor;

xor = bit_and
| xor "^" bit_and;

bit_and = eq
| bit_and "&" eq;

eq = comp
| eq ("==" | "!=") comp;

comp = shift
| comp ("<" | "<=" | ">" | ">=") shift;

shift = add
| shift ("<<" | ">>") add;

add = mult
| add (""+" | "-") mult;

mult = pow
| mult ("*" | "/" | "%") pow;

pow = unary
| pow "**" unary;

unary = primary
| ("-" | "+" | "!" | "~") unary

primary = literal
| id
| func_call
| "(" expr")";

literal = boolean
| numeric
| string
| array;

boolean = “true" | “false";

numeric = decimal
| hexadecimal
| octal
| binary
| float
| e_notation;

decimal = digit+;
hexadecimal = "0x" hex_digit+;
octal= "0o" oct_digit+;
binary = "0b" bin_digit+;
float = digit+ "." digit+ | "." digit+;
e_notation = (digit+ | float) [eE] [+-]? digit+;
digit = "0"..."9";
hex_digit = digit | "a"..."f" | "A"..."F";
oct_digit = "0"..."7";
bin_digit = "0" | "1";

string = "\'" char* "\'";

func_call = id "(" [expr ("," expr)*] ")";
id = letter (letter | digit | "_")*;
letter = "a"..."z" | "A"..."Z";

array = "[" [expr ("," expr)*] "]";
```

Рис. 8: Грамматика языка

5. Тестирование и внедрение

В данной главе описывается тестирование реализованной библиотеки feev (Fastest Expression EValuator — эвалюатор с компиляцией в замыкания), сравнение с аналогами и интеграция в сервисы антибот-системы Ozon.

5.1. Модульное тестирование и сравнительный анализ библиотеки

Каждая часть эвалюатора покрыта юнит-тестами на 85% с использованием различных тест-кейсов, охватывающих граничные случаи. В частности, для библиотеки специально разработаны тесты, проверяющие обработку пустых и некорректных данных, комбинации операторов с разными приоритетами, обработку специальных символов и escape-последовательностей в строковых литералах, корректность типизации и невалидный пользовательский ввод.

Бенчмарки проводились со следующими характеристиками устройства: CPU — Apple M1 Pro, 8 ядер, 3.2 GHz, RAM — 16 GB LPDDR5, macOS 14.4.1.

Разработаны бенчмарки исполнения выражений, сравнивающие feev на основе хеш-таблицы и массива. В качестве тестовых случаев были выбраны граничные ситуации, в основном с переменными, так как акцент был сделан на сравнении работы ValueProvider, который предоставляет доступ к переменным. В тестах использовались выражения, включающие константы и переменные, тернарные операторы с множественными вызовами функций, а также сложные выражения с сотнями логических операций и большими массивами (порядка ста элементов), переменными и длинными строками, длина которых могла достигать 100 символов. Как показали результаты, при использовании слайсов ValueProvider демонстрирует повышение производительности в 1,5-2 раза в сравнении с использованием хеш-таблиц, что видно в прилагаемой таблице 1.

Test Case	**Time (ns/op)**
Simple expr	
map	165.0
func_slice	85.30
Multi_call contains func	
map	214.7
func_slice	155.2
Complex expr	
map	3093.0
func_slice	1918.0

Таблица 1: Сравнение Value Provider на основе хеш таблицы и массивом функций-значений

В таблице ниже представлено сравнение feev с аналогами, как Expr и Govaluate. Как видно, feev не аллоцирует память во время исполнения, что приносит несколько значительных преимуществ, включая снижение нагрузки на систему за счет минимизации операций выделения памяти, повышение общей производительности в условиях высокой частоты выполнения выражений. В выражениях, где отсутствуют переменные и функции, feev лишь прокидывает готовые значения через замыкания, что ещё больше ускоряет процесс исполнения. В более сложных выражениях, содержащих многочисленные вызовы функций и переменные, feev демонстрирует значительно лучшие результаты, что можно увидеть в таблице 2, обеспечивая до 30 раз большую скорость в сравнении с другими бибоиотеками, что особенно заметно при большом объеме выделяемой памяти.

Test Case	Time (ns/op)	Memory (B/op)	Allocations/op
Simple expr without vars			
feev	0.9477	0	0
expr	46	0	0
goval	377	56	6
Simple expr			
feev	78	0	0
expr	1478	504	59
goval	4233	3208	356
fibonacci(five)			
feev	22	0	0
expr	79	24	2
goval	157	40	3
Multi_call contains func			
feev	152	0	0
expr	316	96	3
goval	547	184	7
Complex expr			
feev	1661	0	0
expr	46853	19440	1826

Таблица 2: Сравнение feev, expr и govaluate

5.2. Интеграция

Библиотека успешно интегрирована во все 10 агент-сервисов, отвечающих за фильтрацию трафика по правилам, что позволило снизить среднее потребление ресурсов агентами на 359 CPU-ядер и 152 ГБ RAM. Кроме того, среднее время обработки запроса сенсором ручного рейт-лимитера (компонента агента, применяющего правила на основе параметров запроса) сократилось на 8%. Бенчмарки, сравнивающие feev и Expr на реальных данных запросов с использованием 30 правил самого нагруженного агента, подтвердили, что разработанная библиотека в среднем в 2 раза быстрее аналога.

Заключение

В рамках данной работы были выполнены следующие задачи:

1. Определено положение библиотеки исполнения правил в цепочке обработки запросов антибот-системы.
2. Проведен анализ Golang-библиотек Expr и Govaluate, реализующих разные подходы к исполнению правил, с выявлением их недостатков и оценкой альтернативного подхода на замыканиях.
3. Спроектированы ключевые компоненты библиотеки и их функциональные назначения.
4. Реализована спроектированная библиотека исполнения правил на Golang, включающая:
   - Парсер на основе алгоритма Пратта для обработки DSL;
   - Типизированный компилятор, преобразующий выражения в замыкания с устранением избыточных вычислений в момент исполнения;
   - Провайдер переменных на основе хеш-таблицы и массива;
   - Синтаксис языка совместимый с текущими правилами антибот-системы;
5. Библиотека была покрыта модульными тестами на 85%, интегрирована во все сервисы антибот-системы Ozon с ручным рейтлимитером и проведен сравнительный анализ производительности библиотеки с помощью бенчмарков на синтетических и реальных данных, а также метрик с сервисов антибот-системы.

Исходный код библиотеки находится в приватном репозитории компании Ozon, но в ближайшем будущем планируется выложить в официальный репозиторий «Ozon Tech»[8].

Список литературы

[1] ClosureRust. — URL: https://blog.cloudflare.com/building-fast-interpreters-in-rust/ (дата обращения: 2025-03-03).

[2] DDoS. — URL: https://s2.ist.psu.edu/paper/DDoS-Chap-Gu-June-07.pdf (дата обращения: 2025-05-06).

[3] Expr. — URL: https://expr-lang.org/docs/language-definition (дата обращения: 2025-03-03).

[4] Golang. — URL: https://go.dev/ (дата обращения: 2025-05-06).

[5] Google. — URL: https://cloud.google.com/ (дата обращения: 2025-03-03).

[6] Govaluate. — URL: https://github.com/Knetic/govaluate/tree/master (дата обращения: 2025-03-03).

[7] Ozon. — URL: https://ru.wikipedia.org/wiki/Ozon (дата обращения: 2025-03-03).

[8] OzonTech. — URL: https://github.com/ozontech (дата обращения: 2025-05-25).

[9] PrattParser. — URL: https://journal.stuffwithstuff.com/2011/03/19/pratt-parsers-expression-parsing-made-easy/ (дата обращения: 2025-03-03).

[10] RateLimiter. — URL: https://developers.cloudflare.com/waf/rate-limiting-rules/ (дата обращения: 2025-05-06).

[11] Uber. — URL: https://www.uber.com/fr/en/ (дата обращения: 2025-03-03).