МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ
БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ФАКУЛЬТЕТ ПРИКЛАДНОЙ МАТЕМАТИКИ И ИНФОРМАТИКИ
Кафедра многопроцессорных систем и сетей

РИСНИК
Павел Станиславович

РАЗРАБОТКА РАСПРЕДЕЛЕННОГО ПРИЛОЖЕНИЯ ИССЛЕДОВАНИЯ СВОЙСТВ ФОРМАЛЬНЫХ ЯЗЫКОВ, ЗАДАННЫХ СИСТЕМОЙ УРАВНЕНИЙ

Дипломная работа

Научный руководитель:
старший преподаватель
В. В. Рябый

Допущена к защите
«___»_____2025 г.
Зав. кафедрой многопроцессорных систем и сетей
кандидат физ.-мат. наук, доцент И.Е.Андрушкевич

Минск, 2025

ОГЛАВЛЕНИЕ

РЕФЕРАТ

Дипломная работа, 56 страниц, 2 рисунка, 5 приложений, 21 источник.

Ключевые слова: МИКРОСЕРВИСНАЯ АРХИТЕКТУРА, КОНТЕКСТНО-СВОБОДНЫЕ ГРАММАТИКИ, МНОЖЕСТВА *FIRST* И *FOLLOW,* СИНТАКСИЧЕСКИЙ АНАЛИЗ *LL(1)* ГРАММАТИК, РАСПРЕДЕЛЕННОЕ ВЕБ-ПРИЛОЖЕНИЕ.

Объекты исследования – формальные языки, распределенные веб-приложения.

Цель работы – изучение алгоритмов нахождения множеств *FIRST* и *FOLLOW* для нетерминалов контекстно-свободных грамматик, задающих формальные языки. Изучение алгоритма нерекурсивного табличного предиктивного синтаксического анализа для грамматик класса *LL(1)*. Изучение принципов проектирования и разработки распределенных веб-приложений и построение микросервисного веб-приложения, предоставляющего реализацию вышеуказанных алгоритмов.

Методы исследования – теоретический анализ и исследование существующих алгоритмов, подходов.

Результатом является реализованное распределенное веб-приложение, которое предоставляет функциональность исследования свойств формальных языков.

Областью возможного практического применения является образовательный процесс в контексте самостоятельного изучения свойств формальных языков.

РЭФЕРАТ

Дыпломная праца, 56 старонак, 2 малюнка, 5 дадаткаў, 21 крыніца.

Ключавыя словы: МІКРАСЭРВІСНАЯ АРХІТЭКТУРА, КАНТЭКСТНА-ВОЛЬНЫЯ ГРАМАТЫКІ, МНОСТВА *FIRST* І *FOLLOW*, СИНТАКСІЧНЫ АНАЛІЗ *LL(1)* ГРАМАТЫК, РАЗМЕРКВАНЫ ВЭБ-ДАДАТАК.

Аб'екты даследвання – фармальныя мовы, размеркаваныя вэб-дадатки.

Мэта працы – вывучэнне алгарытмаў знаходжання мностваў *FIRST* і *FOLLOW* для нетэрміналаў контэкстна-вольных граматык, задаючых фармальныя мовы. Вывучэнне алгарытма нерэкурсівнага таблічнага прэдыктывнага сінтаксічнага аналізу для граматык класа *LL(1)*. Вывучэнне прынцыпаў праектавання и распрацоўцы размеркаваных вэб-дадаткаў і пабудова мікрасэрвіснага вэб-дадатка, які прадастаўляе рэалізацыю вышепаказаных алгарытмаў.

Метады даследвання – тэарэтычны аналіз і даследванне існуючых алгарытмаў, падыходаў.

Вынікам з'яўляецца рэалізаваны размеркаваны вэб-дадатак, які прадастаўляе функцыянальнасць даследвання ўласцівасцей фармальных моў.

Вобласцю магчымага практычнага прымянення з'яўляецца адукацыйны працэс ў кантэксце самастойнага вывучэння ўласцівасцей фармальных моў.

ABSTRACT

Diploma paper, 56 pages, 2 drawings, 5 appendices, 21 sources.

Keywords: MICROSERVICE ARCHITECTURE, CONTEXT-FREE GRAMMARS, FIRST AND FOLLOW SETS, SYNTAX ANALYSIS OF LL(1) GRAMMARS, DISTRIBUTED WEB-APPLICATION.

The objects of the study – formal languages, distributed web-applications.

The purpose of the work – research of the algorithms used to find FIRST and FOLLOW sets for non-terminals of contexts free grammars that define formal languages. Research of the non-recursive predictive table-based algorithm for parsing LL(1) class of grammars. Research of a distributed web-application design and development principles and building a microservice web-application that provides an implementation of the aforementioned algorithms.

Research methods – theoretical analysis and research of existing algorithms, approaches.

The result is an implemented distributed web-application, which provides functionality of research of formal language properties.

The area of possible practical application is an education process in the context of independent study of formal language properties.

ВВЕДЕНИЕ

Формальные языки представляют собой множества слов, заданных над определенным набором символов или алфавитом. Они широко используются, в частности, для задания синтаксиса языков программирования. Зная, какие конструкции могут содержаться в формальном языке, можно определить корректность программы, т.е. определить, содержится ли слово, соответствующее данной программе внутри языка.

Одним из способов задания формального языка, является определение формальной грамматики, через которую могут быть получены слова, соответствующие только данному языку. Исследуя свойства формальных грамматик, мы исследуем свойства формальных языков, которые заданы этими грамматиками.

В данной работе исследуются контекстно-свободные грамматики, рассматриваются фундаментальные множества *FIRST, FOLLOW* для каждого из нетерминалов грамматик, исследуются грамматики класса *LL(1)*, алгоритм синтаксического анализа грамматик класса *LL(1)*.

Разрабатываемое в данной работе приложение, представляет из себя распределенную микросервисную систему, которая, в силу своих характерных свойств, предоставляет возможности точечной масштабируемости, отказоустойчивости, независимого развертывания, распределения нагрузки, технологической гетерогенности.

Исследование принципов микросервисной архитектуры и распределенных систем в целом, а также инструментов построения соответствующей инфраструктуры, также занимает значительную часть данной работы.

Целью данного проекта является разработка распределенного приложения исследования свойств контекстно-свободных грамматик.

Для осуществления поставленной цели, требуется решить следующие задачи:

– исследовать принципы построения распределенных микросервисных систем;

– исследовать такие свойства контекстно-свободных, как множества *FIRST, FOLLOW* над множествами нетерминалов и реализовать алгоритмы нахождения данных множеств;

– исследовать класс грамматик *LL(1)* и реализовать алгоритм синтаксического анализа языка, порождаемого произвольной грамматикой из класса *LL(1)*;

– исследовать и применить современные технологии для построения распределенного приложения, реализующего вышеуказанные алгоритмы.

ГЛАВА 1 РАСПРЕДЕЛЕННАЯ МИКРОСЕРВИСНАЯ АРХИТЕКТУРА СЕРВЕРНЫХ ПРИЛОЖЕНИЙ

1.1 Понятие распределенной системы

Распределенная система – это совокупность автономных вычислительных узлов, соединенных между собой и функционирующих как единая связная система [16].

В зависимости от того, что разделяют между собой узлы распределенной системы, можно выделить системы с разделяемой оперативной памятью, системы с разделяемыми постоянными запоминающими устройствами, системы, не разделяющие между собой ни оперативной памяти, ни постоянных запоминающих устройств [17].

Распределенные веб-приложения, построенные по принципам микросервисной архитектуры, которая будет рассмотрена далее, используют подход, не происходит аппаратного разделения запоминающих устройств. При этом внутри такой распределенной системы может быть как одно общее разделяемое хранилище данных, так и несколько отделенных по бизнес-границам хранилищ. Главное при этом, что хранилища данных предоставляют доступ к данным на уровне приложения, а не на аппаратном уровне.

Распределенные системы способны решать проблемы неэффективной масштабируемости, отказоустойчивости, высокой доступности, надежного хранения данных, непрерывного развертывания и интеграции. При этом, их использование влечет за собой ряд других проблем, таких как обеспечение согласованности и транзакционной семантики, сложность тестирования и мониторинга [2].

1.2 Понятие микросервисной архитектуры

Микросервисная архитектура — это подход к разработке единого приложения, в виде набора небольших сервисов, каждый из которых запущен в своем собственном процессе (или нескольких процессах, например, база данных и сам сервис) и использует облегченные механизмы коммуникации, зачастую, *API HTTP* ресурсов. Микросервисы выстроены вокруг бизнес-возможностей, они могут быть развернуты независимо. Функциональность микросервиса определяется предоставляемым сетевым интерфейсом, поэтому не требуется знание деталей

реализации и возможно использование различных языков программирования и технологий [2].

Снаружи отдельный микросервис рассматривается как черный ящик. Он размещает бизнес-функции в одной или нескольких конечных точках сети по любым наиболее подходящим протоколам. Внутренние детали реализации (например, технология, которая была использована при создании микросервиса, или способ хранения данных) полностью сокрыты от внешнего мира. Изменения, вносимые в пределах границ микросервиса, не затрагивают зависимые системы [2].

Преимущества и недостатки микросервисной архитектуры более подробно будут рассмотрены ниже. Распределенная микросервисная система будет сравниваться с монолитной системой. В монолитной системе, большая часть функциональности, концентрируется в некоторой центральной компоненте. При этом, эта центральная компонента, представляет из себя программу, которая может быть запущена только лишь целиком в одном процессе операционной системы.

1.3 Преимущества микросервисов

1.3.1 Разнородность технологий

Микросервисный подход позволяет использовать разные технологии для отдельных компонент-сервисов, предоставляет свободу подбирать оптимальные инструменты под конкретные задачи. Высокая общая производительность всей системы, может достигаться за счет использования специфичных, наиболее подходящий технологий в соответствующих микросервисах. Например, в тех компонентах, где большую значимость имеет низкая задержка ответа, следует рассмотреть вариант использования языков программирования, в которых отсутствует сборщик мусора, таких как *Rust* или *C/C++*.

Появляется гибкость при выборе способов хранения данных. Например, в социальной сети взаимодействия пользователей удобно хранить в графовой базе данных, которая лучше всего отражает связанность социального графа, а пользовательские сообщения могут располагаться в документно-ориентированном хранилище. Архитектура, формируемая в результате (рисунок 1.1), является разнородной.

Ускоряется внедрение новых технологий, упрощается проверка их применимости на практике.

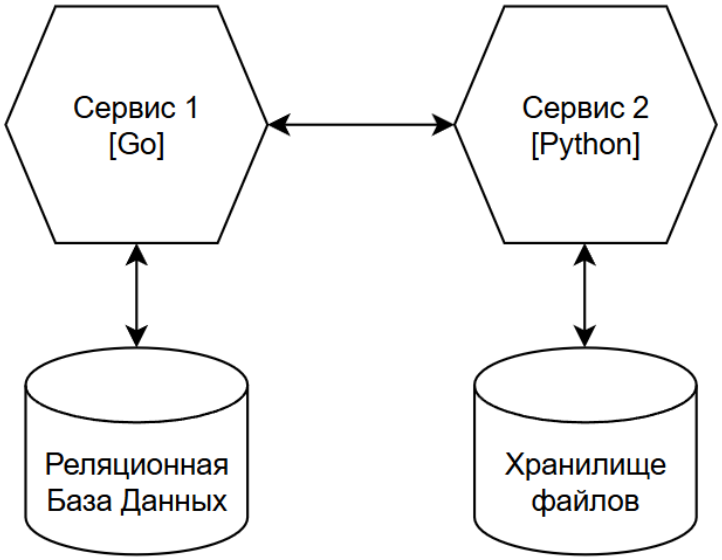


Рисунок 1.1 – Использование разнородных технологий в микросервисах

1.3.2 Отказоустойчивость

Стандартным подходом для обеспечения отказоустойчивости в монолитной архитектуре является развертывание нескольких экземпляров монолита, но в зависимости от размера системы, с этим могут возникать определенные сложности и неудобства.

Микросервисный подход, в силу изолированности микросервисов и их логического разделения по функциональности, дает большую гибкость для реализации механизмов отказоустойчивости. Кроме того, запуск нескольких избыточных копий микросервисов является требует менее мощных аппаратных ресурсов, ведь функциональность системы распределена по нескольким микросервисам.

1.3.3 Точечное масштабирование

При работе с крупным монолитным приложением масштабирование требует развертывания всей системы, даже когда узкое место сосредоточено в отдельном,

независимом модуле внутри приложения. Как правило, подобное использование вычислительных ресурсов не является наиболее эффективным.

Микросервисный подход позволяет точечно масштабировать только те сервисы, которые действительно испытывают повышенную нагрузку. Остальные компоненты системы могут продолжать работать на менее мощном оборудовании, что существенно оптимизирует затраты на инфраструктуру. Как показано на рисунке 1.2, такое избирательное масштабирование дает возможность рационально распределять ресурсы в соответствии с реальными потребностями каждого сервиса.

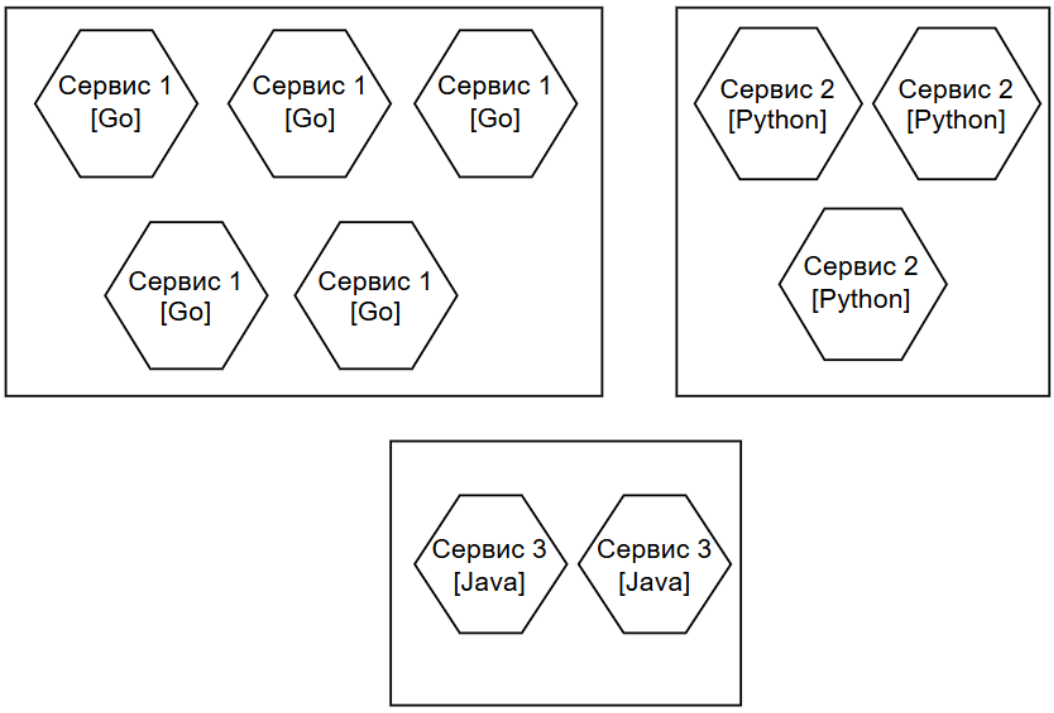


Рисунок 1.2 – Масштабирование приложения по частям

1.3.4 Независимое развертывание

Независимое развертывание – это возможность разворачивать отдельные компоненты системы независимо друг от друга. Данная возможность появляется при соблюдении принципов микросервисной архитектуры, таких как независимость микросервисов, слабая связанность, база данных на микросервис. Незавимое развертывание – одно из главных преимуществ микросервисов, так как

изоляция процесса внесения изменений значительно упрощает процесс исправления ошибок, позволяет чаще выпускать новую функциональность.

1.3.5 Компонуемость

В контексте микросервисной архитектуры под компонентом понимается отдельный микросервис, ведь он может быть использован повторно и развернут изолированно от других частей системы. Важно отметить, что в открытом доступе существует множество бесплатных проверенных высококачественных решений для таких целей как перенаправление запросов, балансировка нагрузки, кеширование, краткосрочное и долгосрочное хранение данных разного рода, анализ данных различного объема, обнаружение сервисов, распределенное отслеживание запросов, мониторинг, агрегирование логов, посредничество при передаче сообщений, оркестрация микросервисов, создание алгоритмов распределенного консенсуса, централизованное хранение информации, потоковая обработка данных, автоматическое масштабирование микросервисов и других.

Компонуемость микросервисов позволяет сосредоточиться на решении конкретных бизнес-задач, и избежать временных трат на вынесение повторяющейся функциональности в отдельные сервисы, что ускоряет процесс разработки и повышает качество всего продукта в целом.

1.4 Недостатки микросервисов

1.4.1 Неудобства при разработке

По мере роста количества сервисов разработчики сталкиваются с рядом сложностей, оказывающих негативное влияние на общую продуктивность. Особенно остро подобные проблемы проявляются при использовании ресурсоемких платформ вроде *JVM*, где ограничения оборудования становятся заметны уже при попытке локального запуска 7-8 сервисов. Ситуация усугубляется при работе с облачными сервисами, которые невозможно, трудно, либо дорого эмулировать в локальном окружении [2].

Опытные команды снижают влияние этих проблем через оптимизацию потребления ресурсов, выбор оптимальных инструментов разработки. Но это никак

не отменяет тот факт, что сложности локальной разработки существуют и негативно влияют на опыт разработчиков [2].

1.4.2 Трудности при составлении отчетности

Монолитные системы, зачастую, используют единую базу данных. В этом случае, проводить анализ данных можно через обращение либо напрямую к этой базе, либо к её копии для чтения.

Микросервисная архитектура предполагает распределение данных между базами данных, соответствующих отдельным сервисам. Для извлечения данных, разбросанных по всей системе, применяется *ETL*-подход (*Extract, Transform, Load*, *ETL*). При нем данные извлекаются из различных источников, затем преобразуются в согласованный формат и после этого загружаются в некоторое централизованное хранилище [10].

Одним из вариантов реализации *ETL*-подхода, является пакетная обработка данных. При пакетной обработке, идет накопление данных за определенных промежуток времени. Затем происходит опциональная трансформация и передача накопленных данных. Переданные данные могут быть эффективно проанализированы, например, через программную модель *MapReduce*. Пакетная обработка экономична с точки зрения ресурсов, но не подходит в ситуациях, когда требуется немедленный доступ к актуальной информации. Примерами технологий пакетной обработки данных являются *Apache Spark, Apache Flink* и другие [10].

Другой вариант реализации *ETL*-подхода – это потоковая обработка данных. Потоковая обработка подходит для задач, где важна минимальная задержка. При данном подходе информация преобразуется и поступает в централизованное хранилище в режиме реального времени. Примерами технологий для потоковой обработки данных являются *Apache Spark, Apache Flink, Kafka Streams* и другие [10].

Таким образом, из всего вышесказанного следует, что налаживание процессов формирования отчетности в микросервисных системах – существенно более сложная задача.

1.4.3 Усложнение процесса мониторинга и устранения неполадок

В монолитных системах приложение либо работает, либо нет, а высокая загрузка центрального процессора, зачастую, указывает на определенные

проблемы. Количество отслеживаемых сервисов обычно невелико, поэтому диагностика довольно проста. В микросервисной же архитектуре ситуация во многом меняется: распределенность усложняет диагностику проблем.

Для улучшения качества проводимой диагностики требуется подготовить соответствующую инфраструктуру. Прежде всего, следует настроить эффективный мониторинг, который позволит отслеживать различные показатели, например, доступность каждого сервиса, время обработки запросов, частоту ошибок, объем обрабатываемого трафика. Примерами инструментов для эффективного мониторинга являются *Prometheus*, *VictoriaMetrics* и другие. Для того, чтобы понимать, как каждый запрос перемещается по ряду сервисов и сколько времени занимает его обработка, следует использовать инструменты распределенной трассировки, такие как *Zipkin*, *Jaeger*, *Sentry*, *Grafana Tempo* и другие. Чтобы упростить процесс ведения логов, можно использовать инструменты агрегации, такие как *Grafana Loki*, *Logstash*, *Fluentd* и другие.

Таким образом, перед переходом к написанию самой бизнес-логики, требуется затратить немало времени на инфраструктуру, которая позволит осуществлять мониторинг и устранение неполадок.

1.4.4 Необходимость усиления системы безопасности

В монолитном приложении основная часть информации обрабатывается в рамках единого процесса, что естественным образом ограничивает поверхность атаки (*attack surface*). В микросервисной системе же данные передаются между различными сервисами через сетевые соединения и, следовательно, увеличивается поверхность атаки. Поэтому микросервисная архитектура требует реализации многоуровневых механизмов защиты. Прежде всего необходимо обеспечить сквозное шифрование всего сетевого трафика между сервисами, чтобы исключить возможность перехвата или подмены данных. Другим важным аспектом является обязательная аутентификация и авторизация всех межсервисных запросов, поскольку в распределенной системе злоумышленник может попытаться выдать себя за того, кем он не является.

Также следует принимать во внимание процесс управления секретами и конфиденциальными данными, ведь их утечка крайне негативно скажется на популярности и востребованности приложения, не говоря уже о возможных проблемах с законодательством. Такие программные продукты, как *HashiCorp*

Vault, *AWS Secrets Manager*, *Azure Key Vault* и другие подобного плана позволяют упростить данный процесс.

Не стоит забывать и об управлении зависимостями. Многие среды разработки, например *InteliJ IDEA*, предоставляют информацию об уязвимостях, ссылаясь на идентификатор уязвимости в программе *CVE* (*Common Vulnerabilities and Exposures Program*). Но прежде всего, важны инструменты анализа композиции программного обеспечения (*Software Composition Analysis*, *SCA*), которые автоматизируют процесс проверки на уязвимости и предоставляют способы смягчения уже имеющихся уязвимостей.

Поэтому, в силу разнообразия технологий и сетевого способа взаимодействия между микросервисами, требуется тратить значительно больше ресурсов для усиления механизмов обеспечения безопасности.

1.4.5 Увеличение задержки

В монолитной системе передача данных между компонентами происходит практически мгновенно, ведь можно использовать общую память. В распределенной архитектуре коммуникация уже не ограничена одним вычислительным узлов. Становятся заметны накладные расходы сетевых задержек. Поэтому хорошей практикой является уменьшение сетевых вызовов всеми возможными способами. Например, если имеется задача извлечь информацию о нескольких объектах по их идентификаторам, можно создать обработчик, который будет доступен через соответствующий *API* и позволит передавать в качестве параметров список идентификаторов объектов. Данный обработчик, позволит снизить сетевые задержки до константы и следственно задержка не будет увеличиваться при необходимости извлечения большего числа объектов. Естественно, речь идет о разумных цифрах, имеющих некоторое ограничение сверху.

Можно рассмотреть варианты объединения нескольких микросервисов в один, чтобы убрать необходимость передачи чего-либо по сети. Однако такой подход не является панацеей, ведь если довести его до крайней степени, то можно получить архитектуру, у которой отсутствуют большинство из преимуществ микросервисов, и при этом имеются недостатки монолита.

1.4.6 Сложности обеспечения согласованности данных

Микросервисных подход усложняет обеспечение согласованности данных. В монолитных системах, где все данные хранятся в единой реляционной базе, можно полагаться на транзакции, имеющие такие свойства как, атомарность, согласованность, изоляция, надежность. Однако при переходе к распределенной архитектуре, где каждый сервис управляет своим состоянием в изолированном хранилище данных, транзакции в прежнем понимании перестают существовать.

Одной из альтернатив транзакциям реляционных баз данных, являются распределенные транзакции, но их реализации, зачастую, имеют ряд своих недостатков, таких как более низкие уровни согласованности, отсутствие сильных уровней изоляции, ухудшение производительности, масштабируемости, доступности. Другие подходы включают в себя использование реализаций шаблона проектирования сага, через использование разнообразных оркестраторов таких, как *Temporal*, *Cadence* и других, или использование более низкого уровня согласованности, если взаимодействие происходит в рамках таких СУБД как *Apache Cassandra, MongoDB.*

Важно тщательно проводить проектирование процесса обработки ошибок, ведь атомарность в распределенном приложении имеет несколько другой характер, отличный от атомарности, предоставляемой транзакциями реляционных баз данных.

Необходимо учитывать возможность наличия повторяющихся сообщений, так как возможны повторные выполнения запросов из-за различного рода сбоев или тайм-аутов. Поэтому, часто требуется обеспечивать идемпотентность обработки сообщений, что создает дополнительные трудности при проектировании и построении системы.

1.5 Вывод

В данной главе были рассмотрены концепции распределенной системы и способа её реализации через использование микросервисной архитектуры. Были исследованы микросервисы, их преимущества и недостатки в сравнении с монолитной архитектурой. Были рассмотрены некоторые шаблоны проектирования распределенных систем, а также приведены примеры бесплатных приложений, нашедших широкое применение в распределенных приложениях серверных систем.

ГЛАВА 2 ФУНКЦИИ *FIRST, FOLLOW*. АЛГОРИТМ ПРЕДИКТИВНОГО СИНТАКСИЧЕСКОГО АНАЛИЗА ДЛЯ *LL(1)* ГРАММАТИК

2.1 Понятие формального языка

Формальный язык — язык, задаваемый формальной грамматикой вида

$$G = (\Sigma, N, S, P) \tag{2.1}$$

которая представляет из себя четверку, где Σ – алфавит, элементы которого называются терминалами, N – множество, элементы которого называют нетерминалами, S – начальный символ грамматики, принадлежащий множеству нетерминалов, из которого могут быть выведены слова, принадлежащие формальному языку, соответствующему грамматике, P – набор правил вывода, или продукций, слов формального языка, соответствующего формальной грамматике, в которых слову состоящему из как минимум одного нетерминального символа и произвольного числа терминальных ставится в соответствие слово, состоящее из произвольного числа терминальных и нетерминальных символов.

Слово принадлежит формальному языку, если оно выводимо из соответствующей формальной грамматики и состоит из терминальных символов.

Контекстно-свободная грамматика — это формальная грамматика, у которой в наборе правил P, вывод слов происходит только из одиночных нетерминалов, т.е. если рассматривать правило, как переход из левой части в правую, то в левой части стоит одиночный нетерминал.

Рассмотрим пример контекстно свободной грамматики для правильных скобочных последовательностей [6]:

$$G = (\{(,)\}, \{S\}, S, P\}) \tag{2.2}$$

где P:

$$\begin{aligned} &S \to (S) \\ &S \to SS \\ &S \to \varepsilon \end{aligned} \tag{2.3}$$

Пример вывода строки «(( ) ( ))» [6]:

$$S \Rightarrow (\,\boldsymbol{S}\,) \Rightarrow (\,\boldsymbol{S}\,S\,) \Rightarrow (\,(\,S\,)\,\boldsymbol{S}\,) \Rightarrow (\,(\,\boldsymbol{S}\,)\,(\,S\,)\,) \Rightarrow (\,(\,)\,(\,\boldsymbol{S}\,)\,) \Rightarrow (\,(\,)\,(\,)\,) \quad (2.4)$$

2.2 Удаление бесполезных нетерминалов из грамматики

2.2.1 Достижимые и недостижимые нетерминалы

Нетерминал A называется достижимым если существует порождение вида

$$S \Rightarrow* \alpha A \beta \quad (2.5)$$

где $\alpha, \beta \in (\Sigma \cup N)^*$, а S – начальный нетерминал, или аксиома [5].

Очевидно, что если нетерминал в левой части правила является достижимым, то и все нетерминалы правой части являются достижимыми. Кроме того, нетрудно заметить, что после удаления из грамматики правил, содержащих недостижимые нетерминалы язык не изменится, то есть множество цепочек, принадлежащих языку, останется прежним.

Алгоритм удаления недостижимых нетерминалов состоит в следующем [5]:

1) добавляем во множество достижимых нетерминалов начальный нетерминал S;

2) проходимся по всем правилам для нетерминалов из множества достижимых, и если в правой части правила находится какой-либо нетерминал, прежде не содержащийся в множестве, то добавляем его;

3) повторяем предыдущий шаг, пока множество достижимых нетерминалов изменяется;

4) берем пересечение от исходного множества нетерминалов и множества достижимых нетерминалов.

Временной функцией сложности вышеприведенного алгоритма, будет функция являющаяся $O(\Gamma^2)$, где Γ – размер грамматики, представляющий собой количество правил, или продукций, имеющихся в грамматике.

Рассмотрим пример применения данного алгоритма для следующей грамматики:

$$G = (\{a\}, \{S, X_1, X_2, X_3, X_4, X_5, X_6, X_7, X_8, X_9\}, S, P) \quad (2.6)$$

где P:

$$
\begin{gathered}
S \to X_1 \\
X_1 \to X_2 \\
X_2 \to X_3 \\
X_3 \to X_4 \\
X_4 \to X_5 \\
X_5 \to X_6 X_7 \\
X_4 \to a \\
X_8 \to X_9
\end{gathered} \tag{2.7}
$$

Применяем алгоритм:

1) добавляем S в множество достижимых нетерминалов, получим $\{S\}$;

2) из S достижим X_1, добавляем его во множество достижимых нетерминалов, получим $\{S, X_1\}$, множество изменилось;

3) из X_1 достижим X_2 добавляем его во множество достижимых нетерминалов, получим $\{S, X_1, X_2\}$, множество изменилось;

4) из X_2 достижим X_3 добавляем его во множество достижимых нетерминалов, получим $\{S, X_1, X_2, X_3\}$, множество изменилось;

5) из X_3 достижим X_4 добавляем его во множество достижимых нетерминалов, получим $\{S, X_1, X_2, X_3, X_4\}$, множество изменилось;

6) из X_4 достижим X_5 добавляем его во множество достижимых нетерминалов, получим $\{S, X_1, X_2, X_3, X_4, X_5\}$, множество изменилось;

7) из X_5 достижимы X_6, X_7 добавляем их во множество достижимых нетерминалов, получим $\{S, X_1, X_2, X_3, X_4, X_5, X_6, X_7\}$, множество изменилось;

8) больше никакие нетерминалы не достижимы, множество не изменилось;

9) берем пересечение исходного множества нетерминалов и множества достижимых нетерминалов, на выходе получим $\{S, X_1, X_2, X_3, X_4, X_5, X_6, X_7\}$.

2.2.2 Порождающие и непорождающие нетерминалы

Нетерминал называется порождающим, если из него может быть выведена конечная терминальная цепочка. Ясно, что по определению языка грамматики, после удаления правил, содержащих непорождающие нетерминалы, язык не изменится, ведь непорождающие нетерминалы не могли участвовать в выводе какого-либо слова.

Алгоритм удаления непорождающих нетерминалов состоит в следующем [5]:

1) берем пустое множество порождающих нетерминалов;

2) проходимся по правилам и рассматриваем их правые части;

3) если правая часть продукции не содержит нетерминалов или содержит только нетерминалы, находящиеся во множестве порождающих нетерминалов, то добавляем нетерминал из левой части в множество порождающих нетерминалов;

4) повторяем предыдущие два шага, пока множество порождающих нетерминалов изменяется;

5) берем пересечение от исходного множества нетерминалов и множества порождающих нетерминалов.

Временной функцией сложности вышеприведенного алгоритма, будет функция являющаяся $O(\Gamma^2)$, где Г – размер грамматики, представляющий собой количество правил, или продукций, имеющихся в грамматике.

Рассмотрим пример для применения данного алгоритма для грамматики (2.6), со множеством продукций (2.7) :

1) берем пустое множество порождающих нетерминалов, получим $\emptyset$;

2) из X_4 достижим терминал a, добавляем X_4 во множество порождающих нетерминалов, получим $\{X_4\}$, множество изменилось;

3) из X_3 достижим X_4, уже находящийся во множестве порождающих нетерминалов, поэтому добавляем X_3 во множество порождающих нетерминалов, получим $\{X_3, X_4\}$, множество изменилось;

4) из X_2 достижим X_3, уже находящийся во множестве порождающих нетерминалов, поэтому добавляем X_2 во множество порождающих нетерминалов, получим $\{X_2, X_3, X_4\}$, множество изменилось;

5) из X_1 достижим X_2, уже находящийся во множестве порождающих нетерминалов, поэтому добавляем X_2 во множество порождающих нетерминалов, получим $\{X_1, X_2, X_3, X_4\}$, множество изменилось;

6) из S достижим X_1, уже находящийся во множестве порождающих нетерминалов, поэтому добавляем S во множество порождающих нетерминалов, получим $\{S, X_1, X_2, X_3, X_4\}$, множество изменилось;

7) больше никакие правила не содержат порождающих нетерминалов, множество не изменилось;

8) берем пересечение от исходного множества нетерминалов и множества порождающих нетерминалов, получим $\{S, X_1, X_2, X_3, X_4\}$.

2.2.3 Полезные и бесполезные нетерминалы

Нетерминал называется полезным в контекстно-свободной грамматике Г, если он может участвовать в выводе, то есть существует порождение вида

$$S \Rightarrow* \alpha A \beta \Rightarrow* w \tag{2.8}$$

где $w \in \Sigma$, $\alpha, \beta \in (\Sigma \cup N)^*$, а S – начальный нетерминал, или аксиома. Иначе он называется бесполезным [5].

Грамматика Г не содержит бесполезных нетерминалов тогда и только тогда, когда грамматика Г не содержит ни недостижимых, ни непорождающих нетерминалов [5].

Докажем достаточность утверждения выше. Рассмотрим любой нетерминал A. Так как он достижим, то $\exists\, \alpha, \beta \in (\Sigma \cup N)^*$, т. ч. $S \Rightarrow* \alpha A \beta$. Кроме того, A является порождающим, т.е. $\exists\, v \in \Sigma^*$, т. ч. $A \Rightarrow* v$. Значит, $\exists\, w \in \Sigma^*, \alpha, \beta \in (\Sigma \cup N)^*$, т. ч. $\alpha A \beta \Rightarrow* w$. Достаточность доказана.

Докажем необходимость утверждения выше. Пусть грамматика содержит недостижимый или непорождающий нетерминал. Но тогда, один или оба вывода из (2.8) не будут существовать. Следовательно терминал будет бесполезным, но по условию грамматика не содержит бесполезных нетерминалов. Получили противоречие. Необходимость доказана.

Алгоритм удаления бесполезных нетерминалов состоит в следующем [5]:

1) удаляем из грамматики правила, содержащие непорождающие нетерминалы;

2) удаляем из грамматики правила, содержащие недостижимые нетерминалы.

После удаления из грамматики правил, содержащих недостижимые нетерминалы, не появятся новые непорождающие нетерминалы [5].

Докажем утверждение выше. Используем доказательство от противного. Пусть, после удаления из грамматики правил, содержащих недостижимые нетерминалы, появился непорождающий нетерминал A. Это значит, что до удаления этих правил, существовало как минимум одно правило, которое делало возможным вывод из A некоторой строки $w \in \Sigma^*$. Возьмем первое из удаленных правил, которое участвовало в выводе строки $w \in \Sigma^*$. Так как оно было удалено, то оно содержало недостижимый нетерминал. Но этот нетерминал не мог быть недостижимым, так используя это правило, его можно было бы вывести из A за 1

шаг, а A, в свою очередь, достижимый нетерминал, так как не был удален. Получили противоречие.

Рассмотрим следующую грамматику:

$$G = (\{a, b\}, \{S, X_1, X_2\}, S, P) \tag{2.6}$$

где P:

$$\begin{gathered} S \to X_1 X_2 a \mid X_2 b \\ X_1 \to a \end{gathered} \tag{2.10}$$

Нетрудно заметить, что если вначале убрать правила, содержащие непорождающие нетерминалы, то получим новый недостижимый нетерминал.

2.3 Система определяющих уравнений и ее решение итерационным методом

Пусть $G = (\Sigma, N, S, P)$, — КС грамматика, $A \in N$. Если множество правил пусто для A, тогда определяющее уравнение для A имеет вид [3]:

$$A = \emptyset \tag{2.11}$$

Пусть $A \to \gamma_1, A \to \gamma_2, \dots, A \to \gamma_m, m \geq 1$, – правила грамматики для нетерминала A, тогда определяющее уравнение для A имеет вид [3]:

$$A = \gamma_1 \mid \gamma_2 \mid \dots \mid \gamma_m \tag{2.12}$$

Совокупность определяющих уравнений всех нетерминалов образуют систему определяющих уравнений грамматики G. Правые части уравнений являются ограниченными регулярными выражениями. Этим выражениям приписываются значения в стандартной интерпретации. Нетерминалы как переменные системы уравнений принимают в качестве значений языки над основным алфавитом Σ [3].

Решением системы уравнений является набор языков, подстановка которых вместо соответствующих нетерминалов, превращает систему уравнений в систему тождеств [3].

Пусть $N = \{ A_i | 1 \leq i \leq n\}$ и $\boldsymbol{A} = (A_1, A_2, \ldots, A_n)$ — вектор-строка нетерминалов, тогда систему определяющих уравнений можно записать в компактной форме [3]:

$$A_i = E_i(\boldsymbol{A}), 1 \leq i \leq n \tag{2.13}$$

где $E_i(\boldsymbol{A})$ — ограниченное регулярное выражение, определяемое правилами нетерминала A_i [3].

В векторной форме система принимает вид [3]:

$$\boldsymbol{A} = E(\boldsymbol{A}), E(\boldsymbol{A}) = \left(E_1(\boldsymbol{A}), E_2(\boldsymbol{A}), \ldots, E_n(\boldsymbol{A})\right) \tag{2.14}$$

Определим рекуррентную систему соотношений на основе определяющей системы уравнений [3]:

$$\begin{gathered} A_i^{l+1} = E_i(\boldsymbol{A}^l) \mid A_i^l , \\ \boldsymbol{A}^0 = (\emptyset, \emptyset, \ldots, \emptyset), \\ \boldsymbol{A}^l = \left(A_1^l, A_2^l, \ldots, A_n^l\right), 1 \leq i \leq n, l \geq 0 \end{gathered} \tag{2.15}$$

В векторной форме рекуррентная система принимает вид [3]:

$$\boldsymbol{A}^{l+1} = \boldsymbol{E}(\boldsymbol{A}^l) \mid \boldsymbol{A}^l, l \geq 0, \boldsymbol{A}^0 = (\emptyset, \emptyset, \ldots, \emptyset) \tag{2.16}$$

Легко видеть, что для каждых i и l, $l \leq i \leq n$, и $l \geq 0$ язык $A_i^l \subseteq \Sigma^*$ суть множество терминальных цепочек, выводимых из нетерминала A_i, деревья выводов которых имеют высоту $h \leq l$. Очевидно также, что $A_i^l \subseteq A_i^{l+1}$ для всех i и l [3].

Множество A_i^∞ всех терминальных цепочек, выводимых из нетерминала A_i, определяется как [3]:

$$A_i^\infty \stackrel{\text{def}}{=} \bigcup_{l=0}^{\infty} A_i^l \tag{2.17}$$

Подставляя эти множества вместо нетерминалов, непосредственно проверяем, что они доставляют решение системе определяющих уравнений [3].

Важные свойства, на которых строится обоснование алгоритма вычисления функции $FIRST_k(X)$ [3]:

1. Множество $t(A_i)$ всех терминальных цепочек, выводимых из нетерминала A_i, определяется по формуле $t(A_i) = \bigcup_{l=0}^{\infty} A_i^l$.

2. Множества A_i^l суть множества терминальных цепочек, выводимых из нетерминала A_i, деревья выводов которых имеют высоту $h \leq l$, при чем для всех i и l, $1 \leq i \leq n, l \geq 0$, имеет место включение $A_i^l \subseteq A_i^{l+1}$.

3. Для фиксированных i и l, $1 \leq i \leq n, l \geq 0$, множество A_i^l конечно по мощности.

2.4 Нахождение функции *FIRST* на множестве нетерминалов контекстно-свободной грамматики

Пусть $G = (\Sigma, N, S, P)$ — КС-грамматика и $L \subseteq \Sigma^*$, k — фиксированное неотрицательное целое число. Определим функцию $first_k(x)$ над Σ^* [3]:

$$first_k(x) \stackrel{\text{def}}{=} y \tag{2.18}$$

где $y = x$, если $|x| \leq$ k, если же $|x| >$ k, тогда $x = yz$, $|y| = k$. Тогда образ языка L от функции $FIRST_k(X)$ есть [3]:

$$FIRST_k(L) \stackrel{\text{def}}{=} \{first_k(x) \mid x \in L\} \tag{2.19}$$

Для функции $FIRST_k(X)$ имеют место следующие свойства [3]:

1) $FIRST_k(L) = FIRST_k\big(FIRST_k(L)\big), L \subseteq \Sigma^*, FIRST_k(\emptyset) = \emptyset, \forall\, k \geq 0$;
2) $FIRST_k(L_1 \cup L_2) = FIRST_k(L_1) \cup FIRST_k(L_2), \forall\, L_1, L_2 \subseteq \Sigma^*$;
3) $FIRST_k(L_1) \cap FIRST_k(L_2) = \emptyset \Rightarrow L_1 \cap L_2 = \emptyset, \forall\, L_1, L_2 \subseteq \Sigma^*, \forall\, k \geq 1$;
4) $L_1 \subseteq L_2 \Rightarrow FIRST_k(L_1) \subseteq FIRST_k(L_2), \forall\, L_1, L_2 \subseteq \Sigma^*$;
5) $FIRST_k(L_1 L_2) = FIRST_k\big(FIRST_k(L_1) FIRST_k(L_2)\big), \forall\, L_1, L_2 \subseteq \Sigma^*$;

$$L_1 \oplus_k L_2 \stackrel{\text{def}}{=} FIRST_k(L_1 L_2) \tag{2.20}$$

6) $(L_1 \oplus_k L_2) \oplus_k L_3 = L_1 \oplus_k (L_2 \oplus_k L_3), \forall\, L_1, L_2, L_3 \subseteq \Sigma^*$;
7) $(L_1 \cup L_2) \oplus_k L_3 = (L_1 \oplus_k L_3) \cup (L_2 \oplus_k L_3), \forall\, L_1, L_2, L_3 \subseteq \Sigma^*$;
8) $L_1 \oplus_k L_2 = L_2 \oplus_k L_1, \forall\, L_1, L_2 \subseteq \Sigma^*$.

Принципиально алгоритм вычисления функции $FIRST_k(X)$ на множестве нетерминалов N КС-грамматики G прост и состоит из двух шагов [3]:

1) по данной грамматике G выписать систему определяющих уравнений, преобразовать её в рекуррентную систему равенств (2.15);

2) вычислять по формуле (2.5) элементы неубывающей ограниченной сверху последовательности $\{\boldsymbol{A}^l \mid l \geq 0\}$, пока $\boldsymbol{A}^l \neq \boldsymbol{A}^{l+1}$. Наименьшее $l = t$, при котором достигается условие $\boldsymbol{A}^l = \boldsymbol{A}^{l+1}$ означает, что функция $FIRST_k(X)$ на множестве нетерминалов N вычислена и $FIRST_k(\boldsymbol{A}) = \boldsymbol{A}^t$.

Можно расширить определение функции $FIRST_k(\beta)$ на произвольную строку символов грамматики, как множество k символов, с которых начинаются строки, порождаемые β.

2.5 Нахождение функции *FOLLOW* на множестве нетерминалов контекстно-свободной грамматики

Пусть $G = (\Sigma, N, S, P)$, – КС грамматика, $A \in N$. Определим на множестве символов $N \cup \Sigma$ грамматики G функцию $FOLLOW_k(X)$ [3]:

$$FOLLOW_k(X) = \bigcup_{\alpha} FIRST_k(\alpha) \tag{2.21}$$

где α – произвольный правый контекст символа X, для которого существует левый вывод $S \Rightarrow * \omega X \alpha$, $\omega \in (\Sigma \cup N)^*, \alpha \in \Sigma^*$.

Рассмотрим метод вычисления функции $FOLLOW_1(X)$, считая грамматику G корректной, то есть не содержащей бесполезных нетерминалов [3].

Метод, вычисления функции $FOLLOW_1(X)$, предлагаемый далее, использует заранее вычисленные значения функции $FIRST_1(X)$ и, также не зависит от свойств исходной грамматики G. Если G – корректная, тогда $FIRST_1(X)$, и, следовательно, $FOLLOW_1(X)$ находят верные значения. В противном случае результаты, корректно вычисленные, должны быть отвергнуты и необходимо корректировать грамматику G и соответствующую систему определяющих уравнений [3].

Введём символ \$, который будем использовать для нашего удобства, как специальный маркер обозначающий конец строки. Предполагается, что этот символ не относится ни к какой грамматике. Также предполагаем, что каждому $A \in N$ соответствует множество $FOLLOW_1(A) = FOLLOW(A)$.

Чтобы вычислить $FOLLOW(A)$ для всех нетерминалов A, будем выполнять следующие шаги, пока ни к одному множеству $FOLLOW$ нельзя будет добавить ни одного терминала [1, с. 286]:

1. помещаем в $FOLLOW(S)$, где S – начальный символ, а \$ – маркер конца строки;

2. если имеется продукция $A \rightarrow \alpha B\beta$, то все элементы множества $FIRST(\beta)$, кроме ε, помещаем во множество $FOLLOW(B)$;

3. если имеется продукция $A \rightarrow \alpha B$ или $A \rightarrow \alpha B\beta$, где $FIRST(\beta)$ содержит ε, то все элементы из множества $FOLLOW(A)$ помещаем во множество $FOLLOW(B)$.

2.6 Синтаксический анализ. *LL(1)* грамматики

Синтаксический анализ, или разбор (*parsing*), представляет собой выяснение для полученной строки терминалов способа его вывода из стартового символа грамматики. На выходе, синтаксический анализатор может генерировать дерево разбора. Формально для данной контекстно-свободной грамматики дерево разбора представляет собой дерево со следующими свойствами [1, с. 82]:

1. Корень дерева помечен стартовым символом.

2. Каждый лист помечен терминалом или ε.

3. Каждый внутренний узел помечен нетерминалом.

4. Если A является нетерминалом и помечает некоторый внутренний узел, а $X_1, X_2, \dots, X_n$ – метки его дочерних узлов слева направо, то должна существовать продукция $A \rightarrow X_1 X_2 \dots X_n$. Здесь каждой из обозначений $X_1, X_2, \dots, X_n$ представляет собой либо терминальный, либо нетерминальный символ. В качестве частного случая продукции $A \rightarrow \varepsilon$ соответствует узел A с единственным дочерним узлом ε.

Синтаксический анализатор, как правило, получает на вход не текстовый поток, а «объекты токенов». Для разбиения текстового потока на объекты токенов используется лексический анализатор. Вместе с терминальными символами, которые используются при синтаксическом анализе, объекты токенов несут дополнительную информацию в форме значений атрибутов [1, с. 118].

Последовательность входных символов, составляющих отдельный токен, называется лексемой (*lexeme*). Таким образом, можно сказать, что лексический анализатор отделяет синтаксический анализатор от представления токенов в виде лексем [1, с.118].

Если в синтаксическом анализе не используются значения атрибутов, то объекты токенов равносильны терминалам.

Грамматика может иметь более одного дерева разбора для данной строки терминалов. Такая грамматика называется неоднозначной (*ambiguous*). Чтобы

показать неоднозначность грамматики, достаточно найти строку терминалов, которая даёт более одного дерева разбора [1, с. 84].

Рассмотрим метод разбора, называемый «рекурсивным спуском». Он относится к нисходящим методам разбора, в которых построение дерева разбора начинается от корня по направлению к листьям. В процессе разбора используется текущий сканируемый символ входной строки, который часто называется «предсимволом» (*lookahead symbol*).

Анализ методом рекурсивного спуска (*recursive-descent parsing*) представляет собой способ нисходящего синтаксического анализа, при котором для обработки входной строки используется множество рекурсивных процедур и с каждым нетерминалом связана своя процедура. При предиктивном анализе методом рекурсивного спуска текущий сканируемый символ однозначно определяет поток управления в теле процедуры для каждого нетерминала. Последовательность вызовов процедур при обработке входной строки неявно определяет его дерево разбора и при необходимости может использоваться для его явного построения [1, с. 104].

Анализатор, работающий методом рекурсивного спуска, может оказаться в состоянии зацикливания. Такая проблема возникает в леворекурсивных продукциях наподобие

$$expr \rightarrow expr + term \tag{2.22}$$

В них левый символ тела продукции идентичен нетерминалу в заголовке продукции. Левую рекурсию можно устранить, переписав некорректную продукцию. Рассмотрим нетерминал с двумя продукциями:

$$A \rightarrow A\alpha \mid \beta \tag{2.23}$$

Здесь α и β последовательности терминалов и нетерминалов, которые не начинаются с A. Тот же результат может быть достигнуть путем переписывания продукции для A с использованием нового нетерминала R:

$$\begin{aligned} A &\rightarrow \beta R \\ R &\rightarrow \alpha R \mid \varepsilon \end{aligned} \tag{2.24}$$

Нетерминал R и его продукция праворекурсивные, поскольку продукция для R содержит R качестве крайнего правого символа и поэтому дерево разбора растет

вниз вправо. Такой метод устранения левой рекурсии называется непосредственным (*immediate*) устранением левой рекурсии [1, с. 108].

Нисходящий синтаксический анализ можно рассматривать как процесс левого порождения входной строки. На каждом шаге нисходящего синтаксического анализа ключевой проблемой является определение продукции, применимой для нетерминала. Предиктивный синтаксический анализ выбирает корректную продукцию путем предпросмотра фиксированного количества символов входной строки; типичной является ситуация, когда достаточно посмотреть только один (очередной) входной символ [1, с. 282].

Класс грамматик, для которых можно построить предиктивный синтаксический анализатор, просматривающий k символов во входном потоке, иногда называют классом $LL(k)$. Из множеств *FIRST*, *FOLLOW* грамматики можно построить «таблицы предиктивного анализа», которые делают явным выбор продукции при нисходящем синтаксическом анализе. Эти таблицы применяются также и при восходящем синтаксическом анализе [1, с. 283].

В процессе нисходящего синтаксического анализа функции *FIRST* и *FOLLOW* позволяют выбрать применяемую продукцию на основании очередного символа входного потока.

Предиктивные синтаксические анализаторы, то есть синтаксические анализаторы, работающие методом рекурсивного спуска без возврата, могут быть построены для класса грамматик $LL(1)$. Первое «L» означает сканирование входного потока слева направо, второе «L» – получение левого вывода, а «1» – использование на каждом шаге предпросмотра одного символа для принятия решения о действиях синтаксического анализатора [1, с. 288].

Требуется принимать во внимание, что в $LL(1)$ грамматике не может быть ни левой рекурсии, ни неоднозначности [1, с. 288].

Грамматика G принадлежит классу $LL(1)$ тогда и только тогда, когда для любых двух различных продукций $A \to \alpha \mid \beta$ грамматики G выполняются следующие условия [1, с. 288]:

1) $\mathrm{FIRST}(\alpha) \cap \mathrm{FIRST}(\beta) = \emptyset$;

2) если $\beta \Rightarrow* \varepsilon$, то $\mathrm{FIRST}(\alpha) \cap \mathrm{FOLLOW}(A) = \emptyset$. Аналогично, если $\alpha \Rightarrow* \varepsilon$, то $\mathrm{FIRST}(\beta) \cap \mathrm{FOLLOW}(A) = \emptyset$.

2.7 Алгоритм табличного нерекурсивного предиктивного синтаксического анализа *LL(1)* грамматик

Рассмотрим алгоритм построения таблицы M предиктивного синтаксического анализа для грамматики G из класса $LL(1)$, где \$ – вспомогательный символ конца строки, как уже было описано для алгоритма вычисления функции FOLLOW, приведенного в разделе 2.5 [1, с. 290]:

1) обходим все продукции $A \to \alpha$ грамматики G;

2) для каждого терминала a из $\mathrm{FIRST}(\alpha)$ добавляем $A \to \alpha$ в ячейку $M[A, a]$;

3) если $\varepsilon \in \mathrm{FIRST}(\alpha)$, то для каждого b из $\mathrm{FOLLOW}(A)$, добавляем $A \to \alpha$ в $M[A, b]$;

4) если $\varepsilon \in \mathrm{FIRST}(\alpha)$ и $\$ \in \mathrm{FOLLOW}(A)$, добавляем $A \to \alpha$ в $M[A, \$]$;

5) выполняем шаги 1-4 пока не обойдем все продукции;

6) во все ячейки, оставшиеся без продукции, выставляем значение *error*, обозначающее ошибку, либо оставляем их пустыми, принимая во внимание тот факт, что обращение к ним будет означать ошибку.

Рассмотрим алгоритм табличного нерекурсивного предиктивного анализа. Анализируемая строка содержится во входном буфере и за ней следует маркер конца строки. Так как алгоритм нерекурсивный, то будет использоваться явный стек, на дне которого, также будет маркер конца строки, а поверх него начальный нетерминал грамматики S. Кроме того, используется таблица предиктивного синтаксического анализа M, которую можно получить, используя вышеуказанный алгоритм. На вход поступает $w \in \Sigma^*, M$. На выходе получим совокупность шагов, следуя которым можно вывести w либо сообщение об ошибке. Псевдокод для данного алгоритма приведен ниже [1, с. 293].

```
i := 0
x := stack.Peek() // присваиваем x начальный нетерминал
var a Symbol // декларируем переменную a типа Symbol
while x != $ { // выполняем, пока стек не пуст
     a = inputBuffer[i] // кладем в a очередной входной символ
     if x == a {
          stack.Pop()
          i++
     } else if x.IsTerminal() {
          error()
     } else if M[x, a] == error {
```

```
            error()
        } else if M[x, a] == x -> ABC…Z {
            print(x -> ABC…Z)
            stack.Pop()
            stack.Push(Z, …, C, B, A)
        }
        x = stack.Peek()
}
```

2.8 Вывод

В данной главе были рассмотрены контекстно-свободные грамматики, как способ задания формального языка. Были исследованы алгоритмы удаления бесполезных нетерминалов, являющиеся важным шагом, приводящие грамматику в более удобную форму, но при этом не изменяющие язык грамматики. Были рассмотрены множества *FIRST* и *FOLLOW*, заданные на множестве нетерминалов контекстно-свободных грамматик, а также итеративные алгоритмы для нахождения множеств *FIRST* и *FOLLOW*. Кроме того, были исследованы *LL(1)* грамматики, был реализован алгоритм нерекурсивного табличного предиктивного синтаксического анализа для *LL(1)* грамматик, в котором были задействованы множества *FIRST* и *FOLLOW*. Таким образом, были исследованы реализации алгоритмов исследования свойств формальных языков, заданных системой определяющий уравнений относящихся к контекстно-свободным грамматикам, и в особенности к классу *LL(1)* контекстно-свободных грамматик.

ГЛАВА 3 РАЗРАБОТКА ПРОГРАММНОГО ПРОДУКТА

3.1 Технологии, используемые при построении инфраструктуры

3.1.1 Язык программирования *Java*

Java представляет собой язык программирования и платформу вычислений, которая была впервые выпущена компанией *Sun Microsystems* в 1995 г. Технология эволюционировала из скромной разработки до инструмента, который играет серьезную роль в современном цифровом мире, предоставляя надежную платформу для множества сервисов и приложений [4].

Java – язык, широко используемый и востребованный в разработке промышленных приложений. Ключевыми особенностями данного языка и его среды исполнения являются уделение особого внимания безопасности, возможность запуска скомпилированного байт-кода на разных платформах, наличие большого количества библиотек, поставляющихся вместе с самим языком и средой исполнения, относительная простота самого языка, гарантии обратной совместимости, отсутствие необходимости в ручном управлении памятью и связанных с этим ошибок, возможность автоматической компиляции часто вызываемого байт-кода в машинный код и другие.

Стоит отметить наличие большого количества сторонних библиотек и фреймворков для упрощения разработки, в частности, веб-приложений. Один из таких фреймворков – *Spring Framework*. *Spring* состоит из множества частей, которые следует добавлять по необходимости, таких как *Spring Web MVC, Spring Data JPA, Spring Boot, Spring Security* и других. Его использование в значительной степени позволяет ускорить процесс написания микросервисных приложений, настроить средства коммуникации между ними, взаимодействие с хранилищем данных или брокером сообщений. Это дает возможность разработчику сконцентрироваться на реализации необходимой бизнес-логики.

Таким образом, язык *Java* имеет развитую экосистему в виде стандартной библиотеки, разнообразных сторонних библиотек и фреймворков и представляет собой хороший выбор при разработке микросервисных приложений.

3.1.2 Язык программирования *Go*

Go – язык разработанный компанией *Google* и выпущенный в 2012 году. Он хорошо подходит и широко используется для написания микросервисных приложений.

Отличительной особенностью *Go* является его модель многопоточности. *Go* не предоставляет прямого доступа к потокам операционной системы, вместо этого используются горутины. В каждой программе функция *main* также запущена в своей горутине. Управление горутинами и назначение их на конкретные потоки операционной системы, берет на себя специальный планировщик, который можно конфигурировать через определенные параметры запуска. Горутины работают в пользовательском пространстве, а не в пространстве ядра, как потоки операционной системы. Это позволяет тратить меньше памяти, так как потоки операционной системы занимают около *1 MiB*, а минимальный размер горутин не превышает *2 KiB*. Кроме того, переключение горутин занимает меньше времени, так как оно происходит в пользовательском пространстве и не требует использования системных вызовов и переходов в пространство ядра операционной системы, достигаемых через прерывания. Планировщик использует кооперативную многозадачность, с элементами вытесняющей многозадачности, чтобы не допустить возникновения ситуации, называемой активная блокировка (*livelock*).

Также, *Go* позволяет программистам прозрачно пользоваться неблокирующим вводом/выводом, который особенно важен при сетевых коммуникациях, широко используемых в микросервисной архитектуре [8].

Основная область применения *Go* – разработка серверных приложений и облачных технологий. Поэтому для него существует множество фреймворков, но основные подходы к написанию кода обработки запросов во всех из этих фреймворков схожи со стандартной библиотекой и, следовательно, их изучение не требует больших временных затрат. Популярными являются такие фреймворки как *Gin, Chi, Fiber, Echo, FastHTTP* и другие.

Исходя из вышеуказанных преимуществ, можно заключить, что язык программирования *Go* является современным языком для написания микросервисов.

3.1.3 Язык программирования *JavaScript*

JavaScript – язык, использующийся преимущественно для разработки фронт-энд части веб-приложений. Современные браузеры предоставляют возможность использовать данный язык, так как в них встроены движки, обеспечивающие исполнение *JavaScript*-кода, например, в *Google Chrome* таким движком является *V8*.

Стоит отметить, что имеется возможность запуска *JavaScript* кода не только в браузере, но и через среду исполнения «*Node.js*», которая, как и *Google Chrome*, использует движок *JavaScript V8* [21].

Использование *JavaScript* для написания интерфейсов в браузере позволяет внести динамические эффекты на веб-страницу и даже работать с одной и той же веб-страницей без необходимости перехода по нескольким.

При написании браузерных интерфейсов, вместе с *JavaScript* используются такие инструменты как *HTML, CSS* и различные расширения, например *Sass* или *Tailwind CSS*. Базовое владение данными инструментами первично по отношению к знанию языка, так как сам динамизм, привносимый *JavaScript*, опирается на взаимодействие с фундаментальными блоками и концепциями данных инструментов.

Для более быстрого написания программных продуктов на языке *JavaScript*, существует большое количество разнообразных фреймворков, предоставляющие разные по степени влияния на разработку абстракции. Один из таких фреймворков – *React* – используется в данной дипломной работе.

Язык *JavaScript* был выбран в первую очередь из-за того, что его выбор является обязательным при написании одностраничных приложений (*Single-Page Application, SPA*). Кроме него, в данном проекте используются такие технологии, как *Tailwind CSS, React, React Router*.

3.1.4 Платформа контейнеризации *Docker*

Контейнеры – технология, позволяющая изолировать приложения, выполняющиеся в операционной системе, путем их пакетирования с включением всех необходимых зависимостей. При этом они занимают меньше памяти и системных ресурсов, чем виртуальные машины, путем использования общего ядра операционной системы.

В данном проекте используется платформа контейнеризации *Docker*, в ней контейнеры создаются на основе *Docker*-образов (*Docker image*).

Docker образы используют многослойную архитектуру файловой системы. Все слои созданные во время построения образа доступны только для чтения и неизменяемы. При создании контейнера создаётся новый слой, который доступен для записи. Данный слой является уникальным для каждого нового контейнера, но при этом слои из общего образа, на основе которых создаются контейнеры остаются общими и позволяют экономить дисковую память. При этом, при удалении контейнера, его верхний слой, предназначенный для записи, не будучи сохраненным в новом образе, также будет удален [18].

Таким образом Docker-контейнеры используют стратегию копирование-при-записи (*Copy-on-Write*, *CoW*). Данная стратегия позволяет эффективно использовать имеющиеся ресурсы множество раз, но для сценариев, когда требуется осуществлять большое количество операций записи, становится заметным существенная потеря производительности. Для решения этой проблемы, а также для возможности хранения данных вне зависимости от текущего состояния контейнера, существуют такие технологии хранения данных как *Docker volumes*, *bind-mounts*, *tmpfs* [18].

Docker также предоставляет гибкие инструменты для настройки сетевого взаимодействия между контейнерами, которое является предпочитаемым способом коммуникации. Имеются драйверы различных типов, самыми популярными из которых являются типы *host* и *bridge*. Сетевой драйвер *host* позволяет контейнеру разделять пространство портов исходной системы, на которой запущен *Docker*. Данный драйвер может использоваться для общего увеличения производительности и в ситуациях, когда требуется слушать на большом количестве портов. Данные преимущества достигаются главным образом за счет того, что, в случае использования сетевого драйвера *host* не применяется трансляция сетевых адресов (*Network Address Translation*, *NAT*). Сетевой драйвер *bridge* предоставляет возможность контейнерам внутри сети взаимодействовать между собой и при этом быть изолированными от других контейнеров, не подключенных к этой сети.

Таким образом, *Docker* предоставляет большое количество разнообразных инструментов, для разработки программ и пакетирования их вместе со всеми необходимыми зависимостями, для воспроизводимого запуска всех машинах, с установленным *Docker Engine*.

3.1.5 Платформа оркестрации контейнеров *Kubernetes*

Системы оркестрации контейнеризованных приложений позволяют управлять инфраструктурой распределенных приложений более удобным и централизированным образом. В данном приложении используется система *Kubernetes*.

Kubernetes предоставляет возможность настроить следующие механизмы внутри распределенного приложения: обнаружение сервисов и балансировка нагрузки, самовосстановление, управление памятью, автоматизирование внесения и отмены обновлений, управление конфиденциальной информацией и конфигурациями, горизонтальное масштабирование, управление вычислительными ресурсами и другие [7].

Kubernetes был выбран как инструмент предоставляющий обнаружение сервисов, балансировку нагрузки, горизонтальное масштабирование, автоматическое восстановление при сбоях, журналирование подов.

3.1.6 Обнаружение сервисов

Обнаружение сервисов – это шаблон проектирования микросервисной архитектуры, который позволяет микросервисам внутри приложения обращаться друг к другу не по конкретным *IP* адресам, а через задаваемые пользователем соответствия имен и адресов. Существуют разные подходы к обнаружению сервисов, но по своей сути они сводятся к некоторому единому источнику правдивой информации, в виде какого-либо сервера. Взаимодействие с этим сервером может происходить явно и напрямую, как, например, в шаблоне построения распределенных систем называющимся сервер обнаружения, так и проходить неявно, как например при использовании *Kubernetes* или шаблона проектирования сервисная сетка.

В текущем проекте используется механизм обнаружения сервисов, предоставляемый платформой *Kubernetes*. Данная платформа предоставляет абстракцию в виде объекта, называющегося сервисом. При создании конфигурационного файла, описывающего какой-либо сервис, указываются поды, которые будут относится к сервису. Затем, сервис, в зависимости от типа, может получить свой собственный *IP*-адрес, который не будет меняться внутри *Kubernetes*-кластера, пока сохраняется соответствующая метаинформация, в независимости от того, сколько существует подов [14].

3.1.7 Обратный прокси *NGINX Ingress Controller*

Распределенное приложение представляет из себя множество функциональных компонент, взаимодействующих между собой. Обратный прокси или *API*-шлюз, это такая компонента, которая разграничивает, то, что доступно пользователям этого приложения и то, что сокрыто.

Обратный прокси является единой точкой входа во все приложение и содержит в себе отображение публичных маршрутов во внутренние, инструменты балансировки нагрузки, сбора логов, а также в некоторых случаях инструменты прекращения использования протокола шифрования *TLS*, инструменты сбора метрик. Кроме того, для доступа к статическим ресурсам, таким как изображения и также *JavaScript*, *HTML*, *CSS* файлы, также используется выстроенный соответствующим образом в обратном прокси маршрут [20].

В данном приложении использует обратный прокси *NGINX Ingress Controller*, который представляет из себя реализацию *Ingress Controller*, приложения внутри кластера *Kubernetes*, которое позволяет ресурсу *Ingress* функционировать [20].

3.1.8 Хранилище данных *MongoDB*

Распределенное хранилище данных – хранилище данных позволяющее хранить данные в виде избыточных копий, или разделенными на непересекающиеся множества по какому-либо признаку, или же используя два вышеуказанных механизма вместе. Хранение данных в виде избыточных копий на нескольких экземплярах хранилища называется репликацией. Репликация позволяет обеспечить отказоустойчивость и высокую доступность.

В данном проекте используется документно-ориентированное хранилище данных *MongoDB*. Оно предоставляет механизмы репликации данных в кластере экземпляров *MongoDB*, в том числе и возможность автоматического аварийного переключения и продолжения работы кластера в случае отказа менее чем кворума узлов. Механизм аварийного переключения заключается в том, что в случае отказа главного узла, его роль на себе берет вторичный узел, обладающий наиболее актуальной копией данных. При этом этот новый главный узел признается главным всеми другими вторичными узлами [19].

MongoDB хранит данные в *JSON* формате, что не всегда позволяет удобно отображать сложные взаимосвязи, но упрощает процесс взаимодействия, если

схема данных имеет связи вида один ко многим, один к одному, или не имеет их вовсе.

Таким образом, отсутствие сложных взаимосвязей между данными, которые подлежат хранению и возможности надежного распределенного хранения, делают *MongoDB* подходящим выбором для данного проекта.

3.1.9 Сервер авторизации *OAuth2 Keycloak*

Сервер авторизации *OAuth2* – специальное приложение, реализующее протокол авторизации *OAuth2* в качестве сервера авторизации. Протокол *OAuth2* четко определяет каким образом происходит процесс аутентификации и авторизации. Через процесс аутентификации клиентское приложение получает определенную строчку, называемую токеном, которая затем используется в *HTTP*-заголовке *Authorization* для аутентификации и авторизации в других приложениях. Эти приложения, выступающие в роли серверов ресурсов, должны будут обратиться на сервер авторизации для проверки действительности и корректности данного токена. Таким образом, логика авторизации и аутентификации вынесена в отдельный компонент, что позволяет не хранить дополнительное состояние внутри микросервисов.

В данном проекте в качестве сервера авторизации, используется приложение с открытым исходным кодом *Keycloak.*

3.2 Разработанные микросервисы

3.2.1 Микросервис исследование свойств контекстно-свободных грамматик

Микросервис исследования свойств контекстно-свободных грамматик, заданных системой определяющих уравнений, предоставляет свою функциональность через конечные точки, работающие по протоколу *HTTP 1.1*. Он позволяет находить форму заданной контекстно-свободной грамматики, в которой она не содержит недостижимых или непорождающих нетерминалов, находить множества *FIRST* и *FOLLOW* для нетерминалов произвольной контекстно-свободной грамматики, проводить табличный нерекурсивный предиктивный

синтаксический анализ для *LL(1)*-грамматик и отображать его шаги. Рассмотрим далее предоставляемую функциональность более подробно.

Микросервис предоставляет следующие конечные точки (endpoints):

1. *POST /processor/retain-generating-nt*.
2. *POST /processor/retain-reachable-nt*.
3. *POST /processor/first1*.
4. *POST /processor/follow*.
5. *POST /processor/parse*.

Для приема *HTTP*-запроса и возврата *HTTP*-ответа на всех вышеуказанных конечных точках используется тип медиа *«Content-type: application/json»*.

На грамматики, принимаемые вышеуказанными конечными точками, накладываются следующие ограничения:

1) грамматики должны принадлежать классу контекстно-свободных грамматик;

2) мощность множества нетерминалов не должна превышать 20;

3) мощность множества терминалов не должна превышать 40;

4) мощность множества правил, содержащихся в определяющих уравнениях, не должна превышать 50;

5) символ «_» обозначает пустой символ, не принадлежащий объединению множеств терминалов и нетерминалов входных грамматик;

6) символ «*__$*» является зарезервированным и, следовательно, не может принадлежать объединению множеств терминалов и нетерминалов входных грамматик;

7) пустая строка либо строка, полностью состоящая из пробельных символов, не может являться символом терминала либо нетерминала;

8) иные ограничения, связанные с форматом передаваемых данных.

В случае нарушения вышеуказанных ограничений, возвращается сообщение об ошибке, содержащее время, когда ошибка была зафиксирована на сервере по нулевому смещению от временного стандарта *UTC*, сообщение об ошибке, и, опционально, список уточняющих сообщений. Сообщение об ошибке имеет тип медиа *«Content-type: application/json»* и код статуса ответа *400 Bad Request*.

В том случае, если при обработке запроса не происходит никаких ошибок, то в ответе ожидается код статуса *200 Ok*.

Конечная точка *POST /processor/retain-generating-nt* принимает на вход в теле запроса грамматику, содержащую множество нетерминалов, множество терминалов, множество определяющих уравнений, стартовый символ. Функциональность, предоставляемая при вызове данной конечной точки, состоит в

следующем: из переданной грамматики убираются непорождающие нетерминалы, то есть из множества нетерминалов убираются непорождающие нетерминалы, а множество определяющих уравнений модифицируется таким образом, что ни одно правило в каком-либо из определяющих уравнений не содержит убранных непорождающих нетерминалов. Кроме того, если стартовый нетерминал, оказывается непорождающим, то в грамматике, возвращаемой в ответе, он заменяется на нулевое *JSON*-значение *null*. Использующийся алгоритм удаления непорождающих нетерминалов, а также пример его работы были рассмотрены в подразделе 2.2.2 данной дипломной работы. После успешной проверки входных данных и применения алгоритма удаления непорождающих нетерминалов, модифицированная грамматика возвращается в виде *HTTP*-ответа, содержащего множество нетерминалов, множество терминалов, множество определяющих уравнений и стартовый символ.

Конечная точка *POST /processor/retain-reachable-nt* принимает на вход в теле запроса грамматику, содержащую множество нетерминалов, множество терминалов, множество определяющих уравнений, стартовый символ. Данная конечная точка предоставляет следующую функциональность: из переданной грамматики убираются недостижимые нетерминалы, то есть из множества нетерминалов убираются недостижимые нетерминалы, а из множества определяющих уравнений убираются, те, в левой части которых стоит недостижимый нетерминал. Использующийся алгоритм удаления недостижимых нетерминалов, а также пример его работы были рассмотрены в подразделе 2.2.1 данной дипломной работы. После успешной проверки входных данных и применения алгоритма удаления недостижимых нетерминалов, модифицированная грамматика возвращается в виде *HTTP*-ответа, содержащего множество нетерминалов, множество терминалов, множество определяющих уравнений и стартовый символ. Поскольку стартовый символ всегда является достижимым, то в HTTP-ответе он не может содержать *JSON*-значение *null*, ибо не будет убран из множества нетерминалов.

Конечная точка */processor/first1* принимает на вход в теле запроса грамматику, содержащую множество нетерминалов, множество терминалов, множество определяющих уравнений, стартовый символ. Сперва идет проверка удовлетворения вышеуказанных ограничений. В случае успешной проверки, грамматика приводится к виду, удобному для нахождения множеств *FIRST*, соответствующих нетерминалам. Для этого осуществляется удаление бесполезных нетерминалов. Использующийся алгоритм удаления бесполезных нетерминалов был описан в подразделе 2.2.3 данной дипломной работы. После приведения

грамматики в корректный вид, применяется алгоритм нахождения множеств *FIRST,* описанный в разделе 2.4 данной дипломной работы. Затем, в качестве ответа, возвращается *JSON*-объект, содержащий в себе другие *JSON*-объекты, такие как приведенная к корректному виду грамматика, которая была использована для нахождения множеств *FIRST,* и сами полученные множества *FIRST* нетерминалов данной приведенной к корректному виду грамматики.

Конечная точка *POST /processor/follow* принимает на вход в теле запроса грамматику, содержащую множество нетерминалов, множество терминалов, множество определяющих уравнений, стартовый символ. Сперва идет проверка удовлетворения вышеуказанных ограничений. В случае успешной проверки, грамматика приводится к виду, удобному для нахождения множеств *FIRST*, *FOLLOW* соответствующих нетерминалам. Для этого осуществляется удаление бесполезных нетерминалов. Использующийся алгоритм удаления бесполезных нетерминалов был описан в подразделе 2.2.3 данной дипломной работы. После приведения грамматики в корректный вид, применяется алгоритм нахождения множеств *FIRST,* описанный в разделе 2.4 данной дипломной работы. Множества *FIRST* нетерминалов грамматики необходимы в работе алгоритма вычисления множеств *FOLLOW* нетерминалов грамматики. Затем происходит вычисление множеств *FOLLOW* по алгоритму, описанному в разделе 2.5 данной дипломной работы. В качестве ответа, возвращается *JSON*-объект, содержащий в себе другие *JSON*-объекты, такие как приведенная к корректному виду грамматика, которая была использована для нахождения множеств *FIRST, FOLLOW* и сами полученные множества *FIRST, FOLLOW* нетерминалов данной приведенной к корректному виду грамматики.

Конечная точка *POST /processor/parse* принимает на вход в теле запроса грамматику, содержащую множество нетерминалов, множество терминалов, множество определяющих уравнений, стартовый символ и строку текстового ввода, в которой должны содержаться символы из терминального алфавита грамматики, разделенные пробелами и/или пробельными символами. На входящую грамматику накладывается дополнительное требование: она должна принадлежать классу *LL(1)*. В том случае, если требование не удовлетворено, выводится соответствующее сообщение об ошибке, в том же формате, как и для ограничений, описанных в начале данного подраздела. Но перед проверкой удовлетворения данному требованию проводится ряд иных действий. Сперва проверяется соблюдение ограничений, указанных в начале данного подраздела. Затем происходит приведение грамматики к корректному виду и нахождение множеств *FIRST, FOLLOW,* аналогично тому, как это было на вышерассмотренной конечной

точке *POST /processor/follow*. Дополнительно, перед нахождением множеств *FIRST* и *FOLLOW*, проводится разбиение на токены входной строки ввода, и проверки значений токенов, имеющихся в тексте, на наличие в алфавите терминалов приведенной к корректному виду входной грамматики. Найденные множества *FIRST, FOLLOW* используются для проверки принадлежности грамматики к классу *LL(1)*, по определению класса грамматик *LL(1)*, имеющемуся в разделе 2.6 данной дипломной работы. После этого, происходит конструирование таблицы предиктивного синтаксического анализа. Для этого используются множества *FIRST, FOLLOW*. Сконструированная таблица затем используется для проведения нерекурсивного предиктивного синтаксического анализа входной строки, разбитой на токены. Использующиеся алгоритм конструирования таблицы и алгоритм табличного нерекурсивного предиктивного синтаксического анализа были рассмотрены в разделе 2.7 данной дипломной работы. В качестве ответа, возвращается *JSON*-объект, содержащий в себе другие *JSON*-объекты, такие как приведенная к корректному виду грамматика и метаданные, полученные в результате проведения табличного нерекурсивного предиктивного синтаксического анализа входной строки, содержащие результат анализа (принадлежит ли входная строка языку, задаваемому входной грамматикой) и информацию о шагах анализа, в которых происходит выбор правила грамматики для терминалов входной строки. Реализованный программных код одного из методов, соответствующих вышеописанным действиям, приведен в приложении А.

При разработке данного микросервиса использовался язык программирования *Java* версии 21 и фреймворк *Spring Boot* [15]. Микросервис состоит из двух слоев: слоя контроллеров и слоя сервисов. Слой контроллеров используется для обеспечения функциональности сетевого взаимодействия. В данном микросервисе имеется один контроллер, предоставляющий вышеуказанные конечные точки. В качестве веб-сервера используется *Embedded Tomcat*, предоставляемый библиотекой *Spring Boot Starter Web*. Слой сервисов представляет из себя набор классов, реализующих алгоритмы исследования свойств формальных языков, которые были рассмотрены в главе 2. Модель данных, которыми оперирует слой контроллеров, отличается от модели данных слоя сервисов. Такой подход используется для уменьшения связанности между двумя слоями. Для отображения одной модели данных в другую используется библиотека *MapStruct* [11], которая позволяет по описанию интерфейсов генерировать высокопроизводительные классы, осуществляющие отображения.

3.2.2 Микросервис предоставления примеров контекстно-свободных грамматик

Микросервис предоставления примеров контекстно-свободных грамматик предоставляет свою функциональность через конечные точки, работающие по протоколу *HTTP 1.1*. Он позволяет получать примеры контекстно-свободных грамматик, заготовленных администратором. Администратор, кроме того, может добавлять новые примеры, или удалять уже имеющиеся. Данный микросервис взаимодействует с хранилищем данных при извлечении, сохранении и удалении примеров контекстно-свободных грамматик.

Микросервис предоставляет следующие конечные точки:

1) *GET /examples*;
2) *GET /examples/{id}*;
3) *POST /examples*;
4) *DELETE /examples/{id}*.

В случае несоблюдения ограничений, связанных с форматом передаваемых данных, или иных ошибках, возвращается сообщение об ошибке, содержащее время, когда ошибка была зафиксирована на сервере по нулевому смещению от временного стандарта *UTC*, сообщение об ошибке, и, опционально, список уточняющих сообщений. Сообщение об ошибке имеет тип медиа *«Content-type: application/json»* и код статуса ответа *400 Bad Request*.

Тела *HTTP*-запросов (ответов), принимаемые (возвращаемые) данным микросервисом имеют тип медиа *«Content-type: application/json»*.

Примеры контекстно-свободных грамматик, возвращаемые и принимаемые данным микросервисом для каждой из грамматик содержат информацию об уникальном идентификаторе, множестве нетерминалов, множестве терминалов, множестве определяющих уравнений, стартовом символе.

Конечная точка *GET /examples* используется для получения страницы с примерами контекстно-свободных грамматик. Без указания каких-либо параметров *HTTP*-запроса запрашивается страница под номером 0, имеющая 3 записи. Грамматики возвращаются в отсортированном по идентификаторам порядке. Порядок сортировки – по возрастанию. Для изменения номера запрашиваемой страницы, требуется указать значение для параметра запроса *page*, которое должно быть неотрицательным. Для изменения количества записей, имеющихся на одной странице, требуется указать значение для параметра запроса *size*, которое должно быть положительным и не превышать 100. Постраничная выдача используется для снижения и ограничения нагрузки, оказываемой на хранилище данных. Данная

конечная точка не требует прохождения аутентификации. В качестве ответа, возвращается информация о текущей странице, вместе со списком соответствующих данной странице контекстно-свободных грамматик. Ожидаемый код статуса ответа – *200 Ok*.

Конечная точка *GET /examples/{id}* используется для получения примера контекстно-свободной грамматики, по её уникальному идентификатору. В случае наличия примера, имеющего указанный идентификатор, пример возвращается в теле ответа, а код статуса ответа имеет значение *200 Ok*. В случае отсутствия примера, возвращается пустое тело ответа и код статуса ответа *404 Not Found*. Данная конечная точка не требует прохождения аутентификации.

Конечная точка *POST /examples* используется администратором для добавления примера контекстно-свободной грамматики. При попытке добавления примера грамматики, с уже имеющимся идентификатором, будет выдано сообщение об ошибке, в том же формате, какой был описан в начале данного подраздела. При успешном добавлении примера грамматики, сообщение ответа будет иметь пустое тело, *HTTP*-заголовок *Location*, в формате */{id}*, где *{id}* – идентификатор добавленного примера, и код статуса ответа *201 Created*. Для вызова данной конечной точке необходимо пройти аутентификацию, посредством предоставления *JSON* веб-токена (*JWT, JSON Web Token*) в *HTTP*-заголовке *Authorization* отправляемого запроса. В том случае, если не удается пройти аутентификацию, возвращается сообщение об ошибке с кодом статуса ответа *401 Unauthorized*.

Конечная точка *DELETE /examples/{id}* используется администратором для удаления примера контекстно-свободной грамматики. При вызове данной конечной точки, ожидается код статуса ответа *204 No Content*. Кроме того, если пример грамматики, под указанным идентификатором, присутствует в базе, то ожидается его удаление. Для вызова данной конечной точке необходимо пройти аутентификацию, посредством предоставления *JSON* веб-токена в *HTTP*-заголовке *Authorization* отправляемого запроса. В том случае, если не удается пройти аутентификацию, возвращается сообщение об ошибке с кодом статуса ответа *401 Unauthorized*.

В качестве хранилища данных, с которым взаимодействует данный микросервис используется документно-ориентированное хранилище *MongoDB*. Оно предоставляет, в частности, механизм репликации данных. В данном случае кластер *MongoDB* состоит из одного главного узла и двух вторичных узлов. Главный узел принимает запросы на запись, отсылает их вторичным узлам и дожидается подтверждения записи на диск от кворума узлов. После необходимого

количества подтверждений он дает возможность клиентам считывать данные с самого себя и отсылает уведомление вторичным узлам, от которых было получено подтверждение, чтобы они тоже сделали видимыми данные, зафиксированные на диске.

В данном микросервисе, хранилище данных настроено таким образом, чтобы предпочитать чтению со вторичных узлов, чтение с главного узла. Чтение с главного узла, в частности, гарантирует чтение недавних записей. Но так как для данного приложения приемлем такой уровень согласованности, как согласованность в конечном итоге, возможность чтения со вторичных узлов остается открытой, что позволяет обеспечить более высокую доступность, ценой потенциальной видимости менее актуальных данных.

При разработке данного микросервиса использовался язык программирования *Java* версии 21 и фреймворк *Spring Boot* [15]. Микросервис состоит из трех слоев: слоя контроллеров, слоя сервисов, слоя репозиториев. Слой контроллеров используется для обеспечения функциональности сетевого взаимодействия. В данном микросервисе имеется один контроллер, предоставляющий вышеуказанные конечные точки. Слой сервисов содержит логику отображения модели данных, использующейся в публичных интерфейсах, в модель данных, использующуюся на уровне репозиториев и работу непосредственно с репозиториями. Для отображения одной модели данных в другую используется библиотека *MapStruct* [11]. В данном микросервисе, используется один сервис, соответствующий сущности. Слой репозиториев представляет собой совокупность репозиториев, которые абстрагируют и инкапсулируют работу с хранилищем данных. В данном микросервисе используется один репозиторий для работы с примером грамматики в качестве сущности. Важно отметить, что реализация репозитория генерируется автоматически, по имеющемуся интерфейсу, так как в данном проекте используется библиотека *Spring Data MongoDB*. Программный код интерфейса данного репозитория, достаточного для генерации автоматической реализации, приведен в приложении Б.

3.2.3 Микросервис предоставления статических ресурсов

Микросервис предоставления статических ресурсов используется для предоставления конечных точки, работающих по протоколу *HTTP*, через которые веб-браузер сможет скачивать *HTML*, *CSS*, *JavaScript* файлы. При запуске данного

микросервиса, необходимо указать директорию, в которой будут находиться файлы, а затем перечислить через пробелы префиксы, которые необходимо будет указывать перед названием непосредственно файлов, для их (файлов) загрузки. Таким образом, список конечных точек определяется динамически. В данном проекте, *Docker*-образ для данного микросервиса настроен таким образом, что конечные точки имеют следующий вид:

– *GET /{pathToFile}*;

– *GET /login/{pathToFile}*.

Вместо *{pathToFile}* подразумевается подстановка пути до файла, находящегося в указанной при запуске директории.

Для оптимизации пропускной способности, при передаче по сети, все файлы дополнительно сжимаются алгоритмом сжатия *Gzip*.

Данный микросервис был разработан с использованием языка *Go* версии 1.24 и фреймворка *labstack/echo/v4* [9].

Программный код реализации данного микросервиса приведен в приложении В.

3.2.4 Микросервис аутентификации пользователей

Микросервис аутентификации пользователей, предоставляет одну конечную точку, использующуюся для аутентификации пользователей. Данная конечная точка, работает по протоколу *HTTP 1.1* и имеет следующий вид: *POST /login*. В качестве тела *HTTP*-запроса принимается *JSON*-объект, содержащий имя пользователя и пароль. В случае успешной аутентификации, в теле *HTTP*-ответа возвращается *JSON* веб-токен (*JWT, JSON Web Token*), а также код статуса ответа *200 Ok*. В случае некорректных данных, возвращается сообщение об ошибке и код статуса ответа *401 Unauthorized*. Полученный токен используется для аутентификации в микросервисе предоставления примеров контекстно свободных грамматик, как было описано выше.

Данный микросервис реализован таким образом, что при обработке запроса, происходит пересылка переданных пользователем данных серверу авторизации *OAuth2 Keycloak*. Было принято решение не организовывать взаимодействие браузерной части приложения с *Keycloak* напрямую, так как согласно протоколу *OAuth2* было бы необходимо использовать веб-интерфейс предоставляемый *Keycloak*.

Данный микросервис был разработан с использованием языка *Go* версии 1.24 и фреймворка *labstack/echo/v4* [9]. Для предоставления конфигурации через текстовые файлы была использована библиотека *joho/godotenv*.

Язык *Go* и его среда исполнения позволяют обеспечить эффективное использование аппаратных ресурсов при сетевом взаимодействии, так как горутины, являющиеся единицами исполнения программного кода, предоставляют неблокирующий ввод/вывод. Это значит, что даже при малом количестве доступных процессорных ядер, микросервис может выдерживать большую нагрузку и своевременно отправлять *HTTP*-ответы.

3.3 Браузерная часть разработанного приложения

Браузерная часть данного приложения была написана с использованием языка *JavaScript*, и фреймворка *React*. Для стилизации веб-страниц использовался *CSS*-фреймворк *Tailwind CSS*. Для маршрутизации в пределах веб-страницы использовалась библиотека *React Router*.

Для взаимодействия с браузерной частью веб-приложения пользователь должен ввести в адресную строку браузера *HTTP*-адрес на котором расположен обратный прокси *NGINX Ingress Controller*, являющийся единой точкой входа в распределенное приложение. В процессе настройки инфраструктуры распределенного приложения был создан ресурс *Kubernetes Ingress*, содержащий правила отображения запрашиваемых адресов к соответствующим адресам микросервисов. В случае если адреса не могут быть явно отображены на адрес каких-либо из имеющихся микросервисов, перенаправление запроса осуществляется в микросервис предоставления статических ресурсов, который предоставляет файлы программного кода *JavaScript*, а также файлы, содержащие *HTML*, и *CSS*.

Непосредственное взаимодействие с другими микросервисами происходит через *AJAX*-запросы (*Asynchronous JavaScript And Xml*), при обращении на *HTTP*-адрес вышеупомянутого обратного прокси и указании соответствующей конечной точки.

Администратор, должен предварительно добавить в адресную строку «*/login*» и пройти процесс аутентификации, чтобы получить возможности удаления и добавления грамматик.

Некоторые фрагменты имеющегося графического интерфейса представлены в приложениях Г, Д.

3.4 Вывод

В данной главе был приведен обзор основных технологий, используемых для построения инфраструктуры микросервисного приложения, таких как языки программирования, контейнеризация, оркестрация контейнеров, обнаружение сервисов, обратный прокси, распределенное хранилище данных, а также обоснован их выбор. Было приведено описание реализованных микросервисов, указано их функциональное назначение, и технологии, использующиеся при реализации. Была рассмотрена реализованная браузерная часть данного распределенного веб-приложения.

ЗАКЛЮЧЕНИЕ

В данной дипломной работе было проведено исследование построения распределенных систем по принципам микросервисной архитектуры, рассмотрены преимущества и недостатки данного подхода к проектированию и разработке программного обеспечения. Было проведено изучение фундаментальных множеств *FIRST, FOLLOW* для нетерминалов контекстно-свободных грамматик, использующихся при построении синтаксических анализаторов грамматик различных классов, в частности для классов *LL(1)*, *LALR(1)*, *SLR(1)*, и алгоритмы вычисления данных множеств. Был реализован алгоритм нерекурсивного табличного предиктивного синтаксического анализа для *LL(1)* грамматик, в котором множества *FIRST* и *FOLLOW* были применены для построения таблицы.

На основании изученного теоретического материала и имеющейся реализации алгоритмов исследования свойств формальных языков заданных системой определяющих уравнений, было построено распределенное одностраничное веб-приложение, запускаемое в оркестраторе контейнеров *Kubernetes* и состоящее из нескольких микросервисов используемых для исследования формальных языков, сервера авторизации *OAuth2*, распределенного хранилища данных *MongoDB*, и других сервисов. Данное приложение предоставляет функциональность исследования свойств формальных языков, заданных системой уравнений.

Результаты дипломной работы могут быть напрямую применены в образовательном процессе, в контексте изучения свойств формальных языков, таких как множества *FIRST* и *FOLLOW* контекстно-свободных грамматик, процесс работы алгоритма нерекурсивного табличного предиктивного синтаксического анализа для грамматик класса *LL(1)*.

Полученные результаты могут быть незначительно доработаны и приспособлены к процессу тестирования знаний студентов, путем автоматического сопоставления значений вышеупомянутых свойств, найденных студентами самостоятельно, со значениями свойств, вычисленными в ходе работы реализованных алгоритмов. Кроме того, результаты дипломной работы допускают расширение, в контексте создания инструмента построения табличных предиктивных синтаксических анализаторов, способных обрабатывать большие грамматики из класса *LL(1)*.

СПИСОК ИСПОЛЬЗУЕМЫХ ИСТОЧНИКОВ

1. Компиляторы: принципы, технологии и инструментарий / А. В. Ахо [и др.]; – 2-е изд. – Москва: Вильямс, 2008. – 1184 с.

2. Ньюмен, С. Создание микросервисов / С. Ньюмен. – Санкт-Петербург : Питер, 2023. – 624 с.

3. Рябый В. В. Методы проектирования лексических и синтаксических анализаторов : учебные материалы для студентов факультета прикладной математики и информатики / В. В. Рябый. – Минск : БГУ, 2015. – 62 с.

4. Справочные ресурсы [Электронный ресурс] // Java. – Режим доступа: https://www.java.com/ru/download/help/whatis_java.html. – Дата доступа: 10.05.2025.

5. Удаление бесполезных символов из грамматики [Электронный ресурс] // ИТМО wiki. – Режим доступа: https://neerc.ifmo.ru/wiki/index.php?title=Удаление_бесполезных_символов_из_грамматики. – Дата доступа: 10.05.2025.

6. Формальные грамматики [Электронный ресурс] // ИТМО wiki. – Режим доступа: https://neerc.ifmo.ru/wiki/index.php?title=Формальные_грамматики. – Дата доступа: 10.05.2025.

7. Что такое Kubernetes [Электронный ресурс] // kubernetes. – Режим доступа: https://kubernetes.io/ru/docs/concepts/overview/what-is-kubernetes/. – Дата доступа: 10.05.2025.

8. Effective Go [Electronic resource] / GO. – Mode of access: https://go.dev/doc/effective_go. – Date of access: 10.05.2025.

9. Introduction [Electronic resource] / echo. – Mode of access: https://echo.labstack.com/docs. – Date of access: 10.05.2025.

10. Kleppmann, M. Designing Data-Intensive Applications / M. Kleppmann. – Sebastopol : O’Reilly Media, 2017. – 640 p.

11. MapStruct 1.6.3 Reference Guide [Electronic resource] / mapstruct. – Mode of access: https://mapstruct.org/documentation/stable/reference/html/. – Date of access: 10.05.2025.

12. Martin Fowler, Microservices [Electronic resource] / M. Fowler // martinFowler.com. – Mode of access: https://martinfowler.com/articles/microservices.html. – Date of access: 10.05.2025.

13. MongoDB Manual [Electronic resource] // MongoDB. – Mode of access: https://www.mongodb.com/docs/manual/. – Date of access: 10.05.2025.

14. Service [Electronic resource] / kubernetes. – Mode of access: https://kubernetes.io/docs/concepts/services-networking/service/#discovering-services. – Date of access: 10.05.2025.

15. Spring Boot [Electronic resource] / spring. – Mode of access: https://docs.spring.io/spring-boot/index.html. – Date of access: 10.05.2025.

16. Steen, M. Distributed Systems / M. Steen, A. S. Tanenbaum. – London : Pearson Education, 2017. – 596 p.

17. Stonebraker, M. The Case for Shared Nothing [Electronic resource] / M. Stonebraker // University of California, Berkley. – Mode of access: https://dsf.berkeley.edu/papers/hpts85-nothing.pdf. – Date of access 10.05.2025.

18. Storage driver [Electronic resource] / dockerdocs. – Mode of access: https://docs.docker.com/engine/storage/drivers/. – Date of access: 10.05.2025

19. Replica Set Elections [Electronic resource] / MongoDB. – Mode of access: https://www.mongodb.com/docs/manual/core/replica-set-elections/. – Date of access: 10.05.2025.

20. The design of NGINX Ingress Controller [Electronic resource] / NGINX Docs. – Mode of access: https://docs.nginx.com/nginx-ingress-controller/overview/design/. – Date of access: 10.05.2025.

21. The V8 JavaScript Engine [Electronic resource] / nodejs. – Mode of access: https://nodejs.org/en/learn/getting-started/the-v8-javascript-engine. – Date of access: 10.05.2025.

ПРИЛОЖЕНИЯ

ПРИЛОЖЕНИЕ А

Программный код одного из методов, использующихся для выполнения синтаксического анализа

```
public ParsingResponse parseInput(ParsingRequest request) {

  Grammar grammar =
toEntityGrammarMapper.toEntity(request.grammarRequest());
  NormalizationService.removeUselessNonTerminals(grammar);

  List<Symbol> tokenizedText = new ArrayList<>(
LexicalAnalyzerService.tokenizeText(request.text(),
grammar.getNonTerminals(), grammar.getTerminals())
);
  tokenizedText.add(Symbol.RESERVED_SYMBOL);

  Map<Symbol, Set<Symbol>> first1 = First1Service.first1(grammar);
  Map<Symbol, Set<Symbol>> follow = FollowService.follow(grammar, first1);
  LL1ParserService.validateThatGrammarIsLL1(first1, follow, grammar);

  Map<LL1ParserService.ParsingTableKey, Word> parsingTable =
LL1ParserService.createPredictiveParsingTable(grammar, first1, follow);

  ParsingMetadata parsingMetadata = LL1ParserService.parseInputUsingTable(
    tokenizedText, grammar, parsingTable
  );

  GrammarResponse grammarResponse = toDtoGrammarMapper.toGrammarResponse(
grammar);
  return ParsingResponse.builder()
           .grammarResponse(grammarResponse)
           .parsingMetadata(parsingMetadata)
           .build();
}
```

Программный код интерфейса репозитория, взаимодействующего с хранилищем данных *MongoDB*

```
package by.bsu.fpmi.grammar.repository;

import by.bsu.fpmi.grammar.entity.Grammar;
import org.springframework.data.domain.Page;
import org.springframework.data.domain.Pageable;
import org.springframework.data.repository.Repository;

import java.util.Optional;

public interface GrammarRepository extends Repository<Grammar, String> {

  Grammar insert(Grammar entity);

  Optional<Grammar> findById(String id);

  void deleteById(String id);

  Page<Grammar> findAll(Pageable pageable);
}
```

ПРИЛОЖЕНИЕ В

Программный код реализации микросервиса предоставления статических ресурсов

```
package main

import (
        "context"
        "fmt"
        "net/http"
        "os"
        "os/signal"
        "syscall"
        "time"
        "github.com/labstack/echo/v4"
        "github.com/labstack/echo/v4/middleware"
        "github.com/labstack/gommon/log"
)

func main() {
        fileDirectory, routes := getRoutingInfo()

        ctx, stop := signal.NotifyContext(context.Background(),
syscall.SIGTERM)
        defer stop()
        server := newStaticServer(routes, fileDirectory)
        logger := server.Logger
        go func() {
           if err := server.Start(":8080"); err != nil && err !=
http.ErrServerClosed {
                 logger.Fatal(err)
           }
        }()
        <-ctx.Done()
        shutdownGracefully(server, logger)
}
```

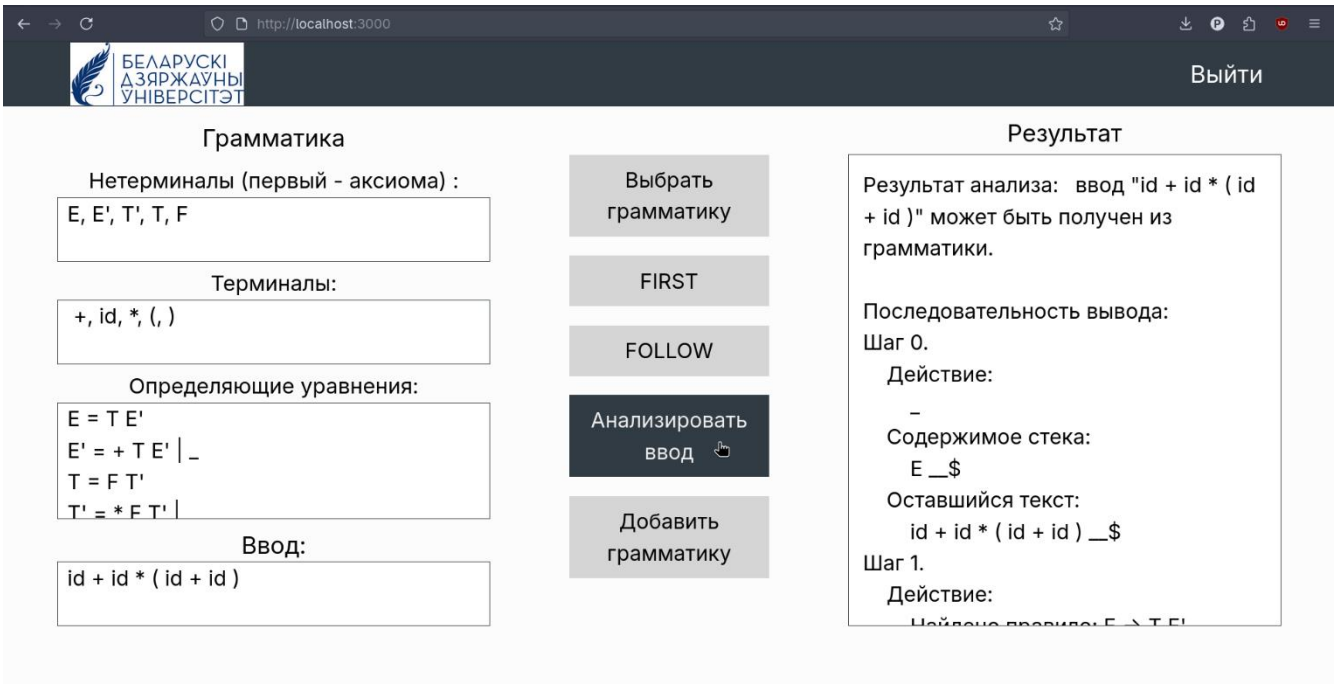
```
func newStaticServer(routes []string, fileDirectory string) *echo.Echo {
        server := echo.New()
        for _, r := range routes {
            server.Static(r, fileDirectory)
        }
        server.Logger.SetLevel(log.INFO)
        server.Pre(middleware.AddTrailingSlash())
        server.Use(middleware.Gzip())
        server.Use(middleware.Logger())
        server.Use(middleware.Recover())
        return server
}

func getRoutingInfo() (string, []string) {
        args := os.Args
        if len(args) < 3 {
            fmt.Println("You should specify file directory as first arg and
then routes, for example: ./static / /login")
            os.Exit(1)
        }
        fileDirectory := args[1]
        routes := args[2:]
        return fileDirectory, routes
}

func shutdownGracefully(server interface{ Shutdown(context.Context) error
}, logger echo.Logger) {
        shutdownCtx, cancel := context.WithTimeout(context.Background(),
3*time.Second)
        defer cancel()
        if err := server.Shutdown(shutdownCtx); err != nil {
            logger.Fatal(err)
        }
        logger.Info("Terminated successfully")
        os.Exit(0)
}
```

ПРИЛОЖЕНИЕ Г

Фрагмент графического интерфейса администратора для главной веб-страницы



ПРИЛОЖЕНИЕ Д

Графический интерфейс веб-страницы входа