**«Санкт-Петербургский государственный электротехнический университет**

**«ЛЭТИ» им. В.И.Ульянова (Ленина)»**

**(СПбГЭТУ «ЛЭТИ»)**

|  |  |  |
| --- | --- | --- |
| **Направление** | 09.03.02 - Информационные системы и технологии | |
| **Профиль** | Информационные системы и технологии в бизнесе | |
| **Факультет** | КТИ | |
| **Кафедра** | АПУ | |
| *К защите допустить* |  | |
| Зав. кафедрой |  | Шестопалов М.Ю. |

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА

БАКАЛАВРА

Тема: Разработка библиотеки парсера на языке libsl для языка С++

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Студент |  |  |  | Ларионов Н.М. |
|  |  | *подпись* |  |  |
| Руководитель | к.т.н., доцент |  |  | Кораблев Ю.А. |
|  | *(Уч. степень, уч. звание)* | *подпись* |  |  |
| Консультанты | ст. преподователь |  |  | Ерошкин А.В. |
|  | *(Уч. степень, уч. звание)* | *подпись* |  |  |
|  | ст. преподователь |  |  | Скрынская О.А. |
|  | *(Уч. степень, уч. звание)* | *подпись* |  |  |
|  | ассистент |  |  | Ряскова Е.Б. |
|  | *(Уч. степень, уч. звание)* | *подпись* |  |  |
|  | к.т.н., доцент |  |  | Белаш О.Ю. |
|  | *(Уч. степень, уч. звание)* | *подпись* |  |  |

Санкт-Петербург

2025

**ЗАДАНИЕ**

**на выпускную квалификационную работу**

|  |  |
| --- | --- |
|  | Утверждаю |
|  | Зав. кафедрой АПУ |
|  | Шестопалов М.Ю. |
|  | «\_\_\_»\_\_\_\_\_\_\_\_\_\_\_\_\_\_20\_\_\_ г. |

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Студент | Ларионов Н.М. | | | |  | Группа | 1371 | |
| Тема работы: Разработка библиотеки парсера для языка LibSL на языке С++. | | | | | | | | |
| Место выполнения ВКР: Санкт-Петербургский государственный электротехнический университет имени В.И. Ленина Исходные данные (технические требования):  ОС: Windows 7 или выше, ОЗУ: 16 Гб, ЦП: 64-разрядный с тактовой частотой 2 ГГц и выше  Содержание ВКР:  Обзор источников, Постановка задачи, Проектирование, Разработка, Тестирование, Экономическое обоснование. | | | | | | | | |
| Перечень отчетных материалов: пояснительная записка, иллюстративный материал | | | | | | | | |
| Дополнительные разделы: Экономическое обоснование ВКР | | | | | | | | |
|  | | | | | | | | |
| Дата выдачи задания | | | Дата представления ВКР к защите | | | | | |
| «1» апреля 2025 г. | | | «19» июня 2025 г. | | | | | |
|  | | |  | | | | | |
| Студент | |  | | Ларионов Н.М. | | | |  |
| Руководитель к.т.н., доцент | |  | | Кораблев Ю.А. | | | |  |
| Консультант ст. преподаватель | |  | | Ерошкин А.В. | | | |  |
| *(Уч. степень, уч. звание)* | |  | |  | | | |  |

**календарный план выполнения**

**выпускной квалификационной работы**

|  |  |
| --- | --- |
|  | Утверждаю |
|  | Зав. кафедрой АПУ |
|  | Шестопалов М.Ю. |
|  | «\_\_\_»\_\_\_\_\_\_\_\_\_\_\_\_\_\_20\_\_\_ г. |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Студент(ка) | Ларионов Н.М. |  | Группа | 1371 |
| Тема работы: Разработка библиотеки парсера для языка LibSL на языке С++ | | | | |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| № п/п | Наименование работ | | | Срок выполнения |
| 1 | Обзор источников | | | 01.04 – 11.04 |
| 2 | Постановка задачи | | | 14.04 – 25.04 |
| 3 | Проектирование библиотеки | | | 28.04 –09.05 |
| 4 | Разработка библиотеки | | | 12.05 – 23.05 |
| 5 | Тестирование библиотеки | | | 26.05 – 30.05 |
| 6 | Создание презентации, прохождение предзащиты | | | 02.06 – 06.06 |
| 7 | Подготовка к защите ВКР | | | 09.06 – 13.06 |
| Студент | | |  | Ларионов Н.М. | | |
| Руководитель к.т.н., доцент | | |  | Кораблев Ю.А. | | |
| *(Уч. степень, уч. звание)* | | |  |  | | |
| Консультант ст. преподаватель | | |  | Ерошкин А.В. | | |

**РЕФЕРАТ**

Пояснительная записка 53 стр., 6 рис., 9 табл., 18 ист.

СПЕЦИФИКАЦИЯ БИБЛИОТЕК, ЯЗЫК СПЕЦИФИКАЦИИ, ПАРСЕР ГРАММАТИКИ, ГЕНЕРАТОР ПАРСЕРОВ

В работе описывается разработка библиотеки парсера грамматики на языке С++ для языка спецификации библиотек LibSL. Рассматривается язык спецификации LibSL и возможности его применения. Описываются подходы к разработке библиотеки парсера, а также процессы проектирования и разработки библиотеки парсера, а также ее тестирования.

В результате выполнения работы была разработана библиотека парсера на языке С++. Использование библиотеки парсера позволяет пользователю отслеживать корректность грамматики содержимого файлов спецификации на языке LibSl, что предоставляет возможность использовать язык LibSL для выполнения различных задач.

ABSTRACT

This work describes development of a of a library of the C++ language grammar parser for the language specifications of LibSL libraries. Also, the work includes the review of specification language LibSL review and the possibilities of this language. The work describes the ways to development of the library parser and processes of designing and developing the library parser with testing include.

As a result of the work, a parser library was developed in the C++ language. Using the parser library allows the user to track the correctness of the grammar of file assessments according to the criteria of the LibSL language, which makes it possible to use the LibSL language to perform various tasks.

содержание

[Определения, обозначения и сокращения 8](#_Toc199884063)

[Введение 10](#_Toc199884064)

[1. Обзор источников 12](#_Toc199884065)

[1.1. Существующие подходы к первичному анализу библиотек 12](#_Toc199884066)

[1.2. Язык спецификации LibSl 12](#_Toc199884067)

[1.3. Аналогичные языки спецификации 14](#_Toc199884068)

[1.4. Существующие решения использования языка LibSl 14](#_Toc199884069)

[2. Постановка задачи 16](#_Toc199884070)

[3. Проектирование 17](#_Toc199884071)

[3.1. Выбор компилятора 17](#_Toc199884072)

[3.2. Выбор автоматизированного средства сборки 18](#_Toc199884073)

[3.3. Выбор формата распространения библиотеки 20](#_Toc199884074)

[3.4. Определение структуры библиотеки 21](#_Toc199884075)

[4. Разработка 25](#_Toc199884076)

[4.1. Подход к разработке 25](#_Toc199884077)

[4.2. Генерация парсера грамматики языка LibSL 25](#_Toc199884078)

[4.3. Разработка библиотеки парсера 26](#_Toc199884079)

[4.4. Сборка библиотеки парсера 33](#_Toc199884080)

[4.6. Подключение библиотеки 38](#_Toc199884081)

[5. Тестирование 40](#_Toc199884082)

[6. Экономическое обоснование 41](#_Toc199884083)

[6.1. Концепция 41](#_Toc199884084)

[6.2. Расходы на оплату труда 41](#_Toc199884085)

[6.3. Отчисления на социальные нужды 44](#_Toc199884086)

[6.4. Материалы 45](#_Toc199884087)

[6.5. Амортизационные отчисления 46](#_Toc199884088)

[6.6 Расчет итоговой себестоимости продукта 48](#_Toc199884089)

[6.7 Вывод 49](#_Toc199884090)

[Заключение 50](#_Toc199884091)

[Список использованных источников 51](#_Toc199884092)

определения, обозначения и сокращения

1. Артефакт, артефакт программирования – экземпляр данных, файл, который был создан в процессе работы программы. Исполняемые файлы программ и библиотек, журналы логов и т.д. являются артефактами.

2. Библиотека – программа, используемая и вызываемая другими программами. Библиотеки не обладают собственной точкой входа в виде главной функции main.

3. Грамматика – формальное описание структуры языка, которое определяет правила синтаксического разбора (парсинга) и лексического анализа (токенизации).

4. Компилятор – программа, переводящая исходный код программы на другой код. В контексте использования С++ подразумевается перевод исходного кода С++ в бинарный машинный код.

5. Лексер – программа, обеспечивающая лексический анализ входных данных, и возвращающая лексические токены [1].

6. Линкер – программа, объединяющая скомпилированные файлы в один бинарный файл или библиотеку.

7. Линковка – процесс объединения скомпилированных файлов в один бинарный файл или библиотеку.

8. ООП – объектно-ориентированное программирование.

9. ОС – операционная система.

10. Парсер – программа, обеспечивающая верификацию и валидацию входных данных, содержащихся в формате лексических токенов [1].

11. Программа – в широком смысле представляет собой последовательность инструкций или машинных команд, выполнение которых компьютером позволяет решать какую-либо задачу. Однако в узком смысле, под программой подразумевается исполняемый файл или его код, который используется человеком. Программа противопоставляется неисполняемым библиотекам, которые могут быть использованы только другими программами, хоть библиотеки и являются программами в широком смысле слова.

12. Репозиторий – хранилище кода и необходимых файлов. Разделяют локальные и удаленные репозитории. Посредством таких удаленных репозиториев, как GitHub и GitLab производится распространение различных проектов, приложений и библиотек.

13. Среда выполнения – вычислительное окружение, обычно библиотека, необходимая для работы компьютерной программы.

14. Среда разработки, IDE – набор инструментов и программ, предоставляемых пользователю для написания кода.

15. Статический анализ кода – вид анализа, который производится без реального выполнения кода, в отличие от динамического анализа кода [1]. Библиотека парсера языка LibSl предназначена для статического анализа кода, написанного на языке спецификации LibSl и содержащегося в файлах с расширением «.lsl».

16. Токен – объект, получаемый после итерации работы лексера. Является минимальной единицей данных, используемой при лексическом анализе данных [1]. Процесс разбиения входных данных на лексические токены называется токенизацией.

17. ЯП – Язык программирования.

18. JVM (Java Virtual Machine) – виртуальная машина, обеспечивающая равнозначное выполнение байт-кода Java, условия выполнения которого не зависит от выбора используемой платформы и операционной системы. Поставляется в составе JDK – Java Development Kit, который помимо виртуальной машине включает в себя компилятор, исполняемый файл Java и библиотеку стандартных классов.

введение

Массовое использование сторонних библиотек является неотъемлемой частью процесса разработки современных программных продуктов [2]. Времени, на разработку актуального продукта в условиях постоянной конкуренции с каждым годом предоставляется все меньше, а количество требований к кибербезопасности, производительности и т.д. становится все больше. В условиях ограниченных временных и человеческих ресурсов, очевидным решением для возможности реализации актуального программного продукта является использование готовых библиотек. Однако, в виду того, что сторонние разработчики зачастую делают исходный код библиотек закрытым, а документация, сопровождающая библиотеки не всегда обладает надлежащим качеством, так как не всегда составляется профильными специалистами, то соответствующее применение библиотек далеко не всегда положительно складывается на техническом качестве программного продукта.

Так как составление классических документов спецификации библиотек имеет свою определенную специфику, то альтернативным подходом является применение языков спецификации [2]. Описание поведения библиотеки при помощи языка спецификации должно являться менее ресурсоемкой задачей, однако чтение и написание такой спецификации будет требовать наличия соответствующих навыков. Помимо этого, полученная спецификация библиотеки может быть использована в различных автоматизированных процессах, вроде поиска ошибок интеграции проекта с библиотекой. Соответственно, для использования языка спецификации необходимы соответствующие программные средства, обеспечивающие работу с файлами спецификации и проверку их грамматики.

Доступность парсера грамматики языка спецификации достаточно сильно зависит от решений, применяющихся для его работы. Так, парсер, написанный на Java будет требовать наличия JVM с подходящей версией, что может вызывать трудности при интеграции парсера с проектом, использующим другую версию JVM или и вовсе другой язык программирования. Расширению доступности применения языка спецификации библиотек LibSl путем разработки библиотеки парсера его грамматики на языке программирования С++ и посвящена данная дипломная работа.

# обзор источников

## 1.1. Существующие подходы к первичному анализу библиотек

Перед тем, как пользователь приступает к подключению и использовании библиотеки, нему требуется получить необходимую информацию об особенностях применения библиотеки. У пользователя есть несколько способов осуществить это, выбрав при этом глубину погружения в контекст использования библиотеки [1]:

* Получение основной информации интерфейсов и классов библиотеки
* Получение информации о зависимости библиотеки от других библиотек
* Получение примеров использования, включая комментарии пользователей

Примером первого подходя являются read-me файлы, которые обычно размещают в корне репозиториев проектов, например распространяющихся через git. Такой способ позволяет пользователю быстро получить краткую концептуальную информацию о структуре библиотеки, однако исключает ознакомление с тонкостями и особенностями использования библиотеки.

В случае, если необходимо получить информацию о некоторых свойствах библиотеки, вроде актуальных версий библиотеки и ее зависимостях, пользователь может обратиться к специализированным интернет-ресурсам, представляющих собой онлайн-репозитории библиотек.

Однако, зачастую пользователю необходимо более глубоко погрузиться в контекст использования библиотеки, и в таком случае приходится обращаться к различным инструментам спецификации, вроде документации или схем и языков спецификации.

## 1.2. Язык спецификации LibSl

Язык LibSl представляет собой язык спецификации, предназначенный для описания поведения библиотек, который, однако, может использоваться и для описания программ и проектов. Язык LibSl описывает структуру библиотеки, сигнатуры и поведение функций интерфейса API, а также задает сценарии использования библиотеки [2]. LibSl описывает поведение классов, используя систему конечных автоматов, где вершины представляют собой состояния классов, а грани представляют собой функции переходов. Далее представлены основные элементы, которые используются в языке LibSl. В Листинге 1.1 указывается информация о версии языка LibSl и версии используемой библиотеки парсера LibSl. В Листинге 1.2 представлено описание типов. В листинге 1.3 представлено описание автомата.

Листинг 1.1 – Информация о LibSl

|  |
| --- |
| libsl "1.0.0";  library LibSLParser version "1.0.0"; |

Листинг 1.2 – Описание типов

|  |
| --- |
| types {  Parser (libsl.parser.Parser);  Token (string);  Bool (bool);  ErrorCode (int32);  } |

Листинг 1.3 – Описание автомата

|  |
| --- |
| automaton libsl.parser.Parser : Parser {  initstate Idle;  state Parsing;  finishstate Completed;  finishstate Error;  var hasError: Bool = false;  var lastErrorCode: ErrorCode = 0;  shift Idle -> Parsing(startParsing);  shift Parsing -> Completed(finishParsing);  shift Parsing -> Error(errorOccured);  shift any -> Idle(resetParser);  fun startParsing(input: Token)  requires notAlreadyParsing: self in Idle;  {  hasError = false;  lastErrorCode = 0;  } |

Продолжение листинга 1.3

|  |
| --- |
| fun finishParsing()  requires parsingInProgress: self in Parsing;  {  hasError = false;  }  fun errorOccured(code: ErrorCode)  requires parsingInProgress: self in Parsing;  {  hasError = true;  lastErrorCode = code;  }  fun resetParser() {  hasError = false;  lastErrorCode = 0;  }  } |

Язык LibSl помимо базового описания поведения библиотеки, может использоваться и в более сложных структурах. Так, при помощи языка LibSl можно наладить автоматизированную миграцию приложений с одной библиотеки на другой [3] или использоваться в статическом анализе с целью обнаружения ошибок интеграции приложений с библиотеками [4].

## 1.3. Аналогичные языки спецификации

Аналогичные языки спецификации были подробно расписаны в статье, посвященной разработке языка спецификации LibSl [2]. По результатам анализа аналогов и были составлены требования, на основе которых был непосредственно разработан язык LibSl. Таким образом, использование языка LibSl в контексте спецификации поведения библиотек полностью оправдано.

## 1.4. Существующие решения использования языка LibSl

Язык LibSl обладает готовыми инструментами для интеграции и использования. Для проверки грамматики файлов, содержащих спецификацию LibSl, существуют парсер в формате библиотеки, написанный на языке Kotlin [2] и плагин парсера для Visual Studio [1], написанный аналогично на языке Kotlin и использующий языковой сервер для проверки грамматики языка LibSl в режиме реального времени.

Для использования готовых решений необходима соответствующая программная инфраструктура – наличие JVM с правильно настроенным окружением и автоматизированными системами сборки, например Gradle и Maven, версии которых должны быть совместимы с версией JVM. Однако, подключение и использование JVM не всегда необходимо, а пользователь, не обладающий навыками работы с языком Java и не знающий особенностей его использования, может столкнуться с трудностями при установке парсера LibSl, решение которых не будет связано непосредственно с навыками знания спецификации и используемого в библиотеке или проекте ЯП.

Одним из доступных для реализации библиотеки парсера языка LibSl языков, является язык программирования C++. Данный ЯП достаточно широко распространен, и многие пользователи обладают достаточным набором знаний, необходимым для его использования. В дополнение к этому, язык C++ поддерживает методы ООП и позволяет реализовать внутреннюю структуру, аналогичную структуре библиотеке языка спецификации LibSl, которая реализована на ЯП Kotlin. В дополнение к этому, ЯП С++ является кроссплатформенным на уровне компиляции, и соответственно использование библиотеки парсера, написанной с использованием предложенного ЯП, будет доступно на различных операционных системах с минимальными доработками библиотеки парсера, включая ОС Windows и дистрибутивы ОС Linux. Таким образом, использование языка С++ для разработки библиотеки парсера языка LibSl оправдано, и существование указанной библиотеки парсера позволит упростить распространение языка спецификации LibSl и расширит спектр решаемых при его помощи задач.

# постановка задачи

На этапе постановки задачи необходимо определиться с основными принципами, которыми будет необходимо руководствоваться при проектировании и разработке библиотеки парсера языка LibSl и корректно расположить их в соответствии с уровнем их приоритета. Основные принципы, отсортированные по мере снижения их важности, начиная с самого приоритетного, перечислены далее:

* Универсальность – так как дипломная работа посвящена расширению возможностей применения языка LibSl, то необходимо выбрать такой набор решений, при котором использование библиотеки парсера будет максимально широко доступно и минимально зависеть от особенностей применяемой операционной системы и среды разработки.
* Простота – для того, чтобы наибольшее количество пользователей могли использовать разработанную в ходе работы библиотеку парсера, необходимо сделать так, чтобы установка и применение библиотеки парсера были максимально упрощены и требовали минимальное количество действий.
* Удобство сопровождения – для большего распространения библиотеки парсера, необходимо спроектировать структуру библиотеки парсера таким образом, чтобы пользователи, которые могли бы в будущем исправлять неточности библиотеки парсера и модифицировать ее в дальнейшем направлении, затрачивали бы минимальное количество усилий для анализа и исправления структуры разработанной библиотеки парсера.

# проектирование

Прежде, чем определить структуру библиотеки парсера и порядок реализации составляющих ее компонентов, необходимо определить набор инструментов, которые будут использоваться для разработки библиотеки парсера.

## 3.1. Выбор компилятора

Как уже было указано, язык С++ является компилируемым языком программирования. В зависимости от используемой ОС и среды разработки, спектр доступных решений может разниться. Далее приведен краткий обзор и анализ наиболее используемых доступных решений.

MSVC – набор инструментов, включающий в себя компилятор, линкер, базовый набор библиотек и т.д., который встроен в среду разработки Visual Studio от Microsoft. MSVC был разработан специально для разработки приложений на ОС Windows и является наиболее подходящим вариантом для данной ОС, не требуя от пользователя тонкой настройки зависимостей и предоставляющий возможность подробной отладки приложения. Однако, данное решение не поддерживает ОС Linux, и таким образом, выбор данного решения противоречит принципу универсальности.

MinGW-w64 – аналогичный набор инструментов. Включает в себя компилятор GCC, изначально разработанный под дистрибутивы ОС Linux, но в дальнейшем адаптированный под ОС Windows, а также сопутствующие линкер, среду выполнения и набор стандартных библиотек. MinGW требует от пользователя определенных знаний особенностей применения языка С++ и имеет некоторые тонкости при настройке приложений для их использования на различных ОС, однако позволяет сделать библиотеку парсера кроссплатформенной и адаптировать ее использование под Windows и Linux.

Clang – это компилятор С++, который способен работать с различными средами выполнения. Данный компилятор является кроссплатформенным и обладает наиболее подробным отладчиком, однако использование данного компилятора подразумевает наличие инструментов линковки и среды выполнения, что возвращает нас к использованию MSVC или MinGW. Данное решение не расширяет кроссплатформенность проектируемой библиотеки парсера и усложняет процесс использования библиотеки, делая библиотеку парсера зависимой от большего числа компонентов и требуя грамотной настройки структуры их взаимодействия. Однако, данное решение может быть применено в будущем при дальнейшей модернизации библиотеки парсера уже на этапе ее сопровождения.

Таким образом, наиболее подходящим решением будет использование MinGW, который позволяет сделать библиотеку парсера универсальной, не будет излишне перегружать ее структуру дополнительными компонентами. Использование данного компилятора позволяет при необходимости перенести библиотеку парсера на ОС Linux, кроме этого, компилятор может быть потенциально изменен на другие.

## 3.2. Выбор автоматизированного средства сборки

Ручное выполнение всех инструкций при каждой сборке бинарного файла библиотеки парсера, линковки и вызовов компилятора представляет собой трудоемкую и рутинную задачу. Для автоматизации подобных задач, были разработаны различные решения в виде средств автоматизированной сборки. Далее приведен краткий обзор и анализ предлагаемых решений.

Make – стандартное средство автоматизированной сборки для дистрибутивов Linux. Make позволяет зафиксировать все инструкции по сборке библиотеки парсера в сборочных файлах, однако каждый сборочный файл необходимо составлять вручную. Данное средство автоматизированной сборки предназначено для Unix-подобных ОС и для его использования на Windows необходимо дополнительно настраивать программное окружение. Использование только make сопряжено с необходимостью выполнения трудоемкой ручной работы, что противоречит принципу простоты устройства библиотеки парсера и удобству сопровождения, и к тому же не полностью универсально.

Ninja – более современная альтернатива Make, также представляющее собой кроссплатформенное средство автоматизированной сборки, доступное как на ОС Windows, так и на дистрибутивах ОС Linux. За счет более продвинутых в технологическом плане решений, Ninja быстрее выполняет указанные в сборочных файлах инструкции, однако не обладает механизмом автоматического вычисления зависимостей. Стоит отметить, что использование средства автоматизированной сборки Ninja подразумевает использование средств генерации сборочных файлов, например CMake.

CMake представляет собой кроссплатформенный инструмент автоматизированной сборки, генерирующий сборочные файлы, предоставляющий возможность настройки выполняемых сборочных процессов и предназначенный для интеграции со средствами автоматизации сборки приложений. CMake имеет встроенную поддержку генерации сборочных файлов для Ninja, однако может работать и с другими средствами сборки, например Make. Таким образом, применение CMake позволяет автоматизировать процесс создания сборочных файлов, упростив сборку библиотеки парсера, и при этом является универсальным кроссплатформенным решением, доступным и для ОС Windows и для дистрибутивов ОС Linux.

MSBuild – официальный инструмент для сборки проектов Visual Studio от Microsoft. Представляет собой средство автоматизации сборки приложений, предназначенный для работы на Windows и имеющий множество передовых решений, вроде поддержки инкрементальной сборки, поддержки многопоточности и других продвинутых возможностей. Однако, данное решение не поддерживает работу на дистрибутивах ОС Linux, что противоречит принципу универсальности.

Таким образом, наиболее подходящим набором решений, является применение средства генерации сборочных файлов CMake вместе со средством автоматизации сборки Ninja. Такое решение позволяет сделать библиотеку парсера кроссплатформенной и упростить процессы ее разработки и сопровождения.

## 3.3. Выбор формата распространения библиотеки

Библиотека представляет собой программу, выполняемую при вызове другой программой, и таким образом, не имеющей входной точки в виде функции main. Для непосредственного использования, библиотека парсера компилируется в так называемые артефакты – бинарные файлы, представляющие собой набор инструкций, зашифрованных в бинарном машинном коде. В ЯП С++ существует несколько форматов распространения бинарных файлов библиотеки парсера – статические и динамические варианты библиотек.

Статические библиотеки линкуются непосредственно с исполняемым файлом приложения, которое использует библиотеку, на этапе компиляции приложения. Исходные файлы библиотеки, предварительно скомпилированные в объектные файлы, объединяются с файлами приложения. Иными словами, бинарный машинный код библиотеки целиком встраивается в бинарный машинный код исполняемого файла приложения. Такой подход удобен с точки зрения простоты выполнения – для использования библиотеки необходим только ее артефакт, который не зависит от других внешних файлов при запуске. Также, во время выполнения приложения, не затрагиваются дополнительные ресурсы, обеспечивающие загрузку необходимых компонентов библиотеки. Однако у такого подхода есть и некоторые недостатки. Так, артефакт библиотеки будет обладать относительно большим размером, что повлечет за собой увеличение размера исполняемого файла. Кроме этого, при обновлении на новую версию библиотеки, пользователю придется скачивать весь файл библиотеки целиком, что может негативно отразиться на удобстве распространения библиотеки в случае, если в момент обновления библиотеки будет большое количество одновременных скачиваний.

Динамические библиотеки вызываются и загружаются приложением во время его выполнения. Такой подход обеспечивает сокращение размера исполняемого файла программы и позволяет использовать один экземпляр библиотеки сразу для нескольких приложений. Однако, для использования динамической версии библиотеки, помимо бинарного файла непосредственно библиотеки, нужно также наличие всех заголовочных файлов, содержащих объявление переменных, функций и методов классов, и файл в формате .lib, который содержит необходимые ссылки, для линковки с компонентами, содержащихся в основном артефакте библиотеки. Это в свою очередь позволяет при обновлении версии библиотеки, скачивать только измененные файлы, что уменьшает общий объем скачиваемого обновления, но делает выполнение артефакта библиотеки зависимым от внешних файлов, вызовы которых будут дополнительно нагружать систему во время выполнения приложения. Кроме этого, после обновления самой библиотеки, не будет необходимости повторно компилировать исполняемый файл приложения. Использование динамической версии библиотеки имеет свои преимущества, однако менее тривиально, чем использование статической версии библиотеки.

Подходы к использованию статической и динамической версии относительно сильно различаются по файловой структуре и набору действий, необходимых для подключения библиотеки, и имеют свои преимущества и недостатки. Однако, в соответствии с поставленными требованиями, было принято решение остановиться на статическом варианте распространения библиотеки парсера, так как такой подход является более простым и универсальным.

## 3.4. Определение структуры библиотеки

Необходимо разработать статическую библиотеку парсера. Для компиляции артефактов библиотеки парсера будет использован компилятор, предлагаемый набором инструментов MinGW. Сборка и линковка библиотеки парсера будет осуществляться при помощи средства автоматизированной сборки Ninja и генератора сборочных файлов CMake. Разработка библиотеки парсера будет производиться на ОС Windows, и первоначальная версия библиотеки парсера будет предназначаться для использования на данной ОС. Исходя из выбранных инструментов, должна быть определена структура библиотеки парсера.

На рисунке 3.1 представлена структура библиотеки парсера. Директория LibSl‑parser содержит директории, в которых хранятся файлы с описанием логики работы библиотеки парсера в формате методов классов. Файл LibSl.cpp описывает методы главного класса парсера, который будут использоваться при вызове класса другими программами при использовании библиотеки парсера. В файле CMakeLists.txt указаны директории, которые содержат необходимые файлы и библиотека парсера, которая будет получена после объединения файлов.

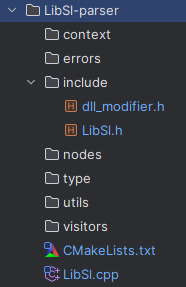


Рисунок 3.1 – Содержание библиотеки парсера

Директория include содержит заголовочные файлы, описывающие объявление классов и функций. Файл LibSl.h описывает главный класс парсера.

На рисунке 3.2 представлена структура директории output, которая предназначена для расположения и хранения собранных экземпляров библиотеки парсера. Папка Debug содержит экземпляры библиотеки парсера, предназначенные для отладки и тестирования. Папка Release содержит рабочие экземпляры библиотеки парсера, которые обладают повышенным быстродействием по сравнению с тестовыми версиями.

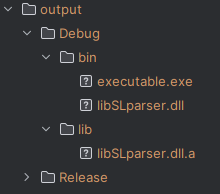


Рисунок 3.2 – Содержание директории output

Так как библиотека парсера не обладает собственной точкой входа в виде главной функции main, то для тестирования библиотеки парсера необходимо ее подключение и использование в тестовом приложении. Общая структура проекта, содержащего библиотеку парсера и тестирующее его приложение представлено на рисунке 3.3.

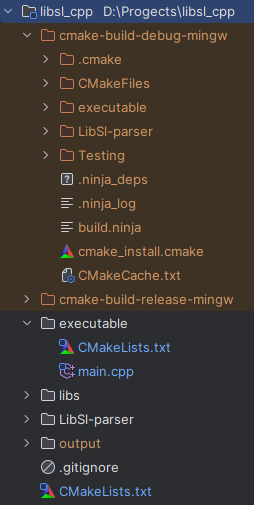


Рисунок 3.3 – Проект, содержащий приложение и библиотеку

Директория проекта содержит в себе директорию executable, в которой содержится файл main.cpp, описывающий функцию, которая производит вызов методов библиотеки парсера с целью тестирования. Директории cmake-build-debug-mingw и cmake-build-release-mingw содержат откомпилированные файлы проекта, которые включают в себя и код тестирующего приложения, и код библиотеки парсера. В директории libs продублированы заголовочные файлы, также содержащиеся в папке include внутри директории библиотеки парсера, и которые необходимы для подключения динамической версии библиотеки парсера к приложению. В корне проекта располагается CMakeLists.txt, который ответственен за сборку проекта, включая приложение и библиотеку парсера.

# разработка

## 4.1. Подход к разработке

Разработка парсера грамматики вручную представляет собой достаточно рутинную и ресурсоемкую задачу [5]. Более универсальным и менее ресурсоемким подходом для разработки парсера грамматики для использования в составе библиотеки будет являться применение автоматизированных средств [6]. Одним из таких решений является ANTLR4. ANTLR4 представляет собой мощный генератор парсеров грамматики, который можно использовать для работы со с структурированным текстом [7]. ANTLR4 позволяет создавать лексеры, парсеры и их комбинации, которые в свою очередь строят на основе полученных данных лексическое дерево и позволяют с ним работать. Что особенно важно, ANTLR4 может генерировать парсеры на различных ЯП, и он поддерживает генерацию парсеров на языке С++ в том числе.

## 4.2. Генерация парсера грамматики языка LibSL

Грамматика языка LibSL описана при помощи языка описания грамматики ANTLR4, что позволяет ее использовать для создания парсера языка LibSL, Парсер языка LibSL, сгенерированный при помощи ANTLR4 состоит из двух частей – лексера и парсера токенов. Упрощенная структура парсера грамматики изображена на рисунке 4.1.



Рисунок 4.1 – Структура сгенерированного парсера

Парсер получает на вход спецификацию, описанную на языке LibSL. Сначала лексер разбивает конструкции языка LibSL на лексические токены или лексемы – минимальные логические единицы языка. После того, как данные на языке LibSL преобразовываются в лексические токены, они передаются парсеру токенов. Парсер токенов проверяет последовательность лексических токенов на предмет их соответствия правилам грамматики языка LibSL, после чего на их основе строит лексическое дерево.

Таким образом, сгенерированный парсер языка LibSL позволяет проверять поступающие на вход данные спецификации, хранящиеся, например, в формате файла или в составе переменной, и выводить положительный или отрицательный результат парсинга, включая ошибки в случае, если они есть.

## 4.3. Разработка библиотеки парсера

Разработанный парсер, выполненный в формате программы, позволяет работать с данными спецификации в различном формате, однако для его работы на компьютере пользователя должны быть установлены необходимые инструменты – компилятор, определенная IDE и соответствующие средства сборки. Кроме этого, особенностью использования ANTLR4 для разработки парсера является то, что для использования сгенерированного парсера необходима среда выполнения, которая обеспечивает его работу.

Для использования программного парсера, каждому новому пользователю придется самостоятельно устанавливать все нужные компоненты и проходить все этапы их сборки. Данный подход может осложняться возможными конфликтами версий у различных обновляемых компонентов и разницей в поведении указанных компонентов на разных платформах и в разных средах.

Таким образом, необходимо внедрить парсер и его компоненты в одну общую структуру библиотеки парсера. В отличии от программной реализации, разработчик библиотеки собирает библиотеку самостоятельно, после чего библиотека распространяется в формате скачиваемых бинарных файлов. В отличии от программной реализации, пользователю для использования парсера в формате библиотеки, достаточно лишь скачать бинарный файл библиотеки правильно подключить его к своей программе, после чего пользователь может использовать библиотеку парсера по своему назначению. Все необходимые компоненты – сгенерированный парсер и среда выполнения, уже будут внедрены в структуру бинарного файла библиотеки. Общая структура библиотеки с необходимыми для работы компонентами изображена на рисунке 4.2.



Рисунок 4.2 – Общая структура библиотеки

Библиотека парсера языка спецификации LibSL состоит из трех основных компонентов – сгенерированного парсера грамматики, который состоит из лексера и парсера токенов, основного класса библиотеки и среды выполнения. Главный класс библиотеки описывает готовые методы, которые позволяют обращаться в необходимом порядке к классам и методам парсера и лексера, сгенерированных при помощи ANTLR4. Также, работу сгенерированного парсера обеспечивает среда выполнения. Необходимые для различных задач методы класса библиотеки, например, парсинг содержимого файла, вызываются внутри программы, к которой подключена библиотека.

В соответствии с поставленной задачей, библиотека парсера должна обеспечивать проверку данных спецификаций на языке LibSL, передаваемых в различных форматах, а именно передаваемых в виде файлов или в составе текстовых переменных. В соответствии с требованиями, был составлен заголовочный файл библиотеки LibSL.h, который описан в листинге 4.1

Листинг 4.1 – Заголовочный файл библиотеки

|  |
| --- |
| #pragma once  #include <string>  #include <memory>  namespace LibSL {  bool parseFromFile(const std::string& filePath);  bool parseFromString(const std::string& input);  std::string getParseTree();  void cleanup();  } |

В заголовочном файле объявляются основные методы библиотеки – парсинг данных спецификации из файла, парсинг данных спецификации из текстовой переменной, вывод лексического дерева и метод освобождения ресурсов памяти. Далее в листинге 4.2 описывается подключение необходимых заголовочных файлов и библиотек, а также инкапсуляция используемых в библиотеке переменных.

Листинг 4.2 – Подключение компонентов и инкапсуляция

|  |
| --- |
| #include "LibSL.h"  #include "LibSLParser.h"  #include "LibSLLexer.h"  #include "antlr4-runtime.h"  #include <fstream>  #include <memory>  #include <sstream>  using namespace antlr4;  namespace {  std::unique\_ptr<ANTLRInputStream> input;  std::unique\_ptr<LibSLLexer> lexer;  std::unique\_ptr<CommonTokenStream> tokens;  std::unique\_ptr<LibSLParser> parser;  LibSLParser::FileContext\* parseTree = nullptr;  } |

В области видимости перечислены переменные – потоки ANTLRInputStream, который передает на вход лексеру поток данных спецификации в формате символов и CommonTokenStream, который передает поток лексических токенов из лексера в парсер токенов. Также указываются классы лексера и парсера токенов, жизненный цикл которых начинается и заканчивается внутри методов библиотеки.

В листинге 4.3 описывается метод библиотеки, выполняющий парсинг данных, передаваемых через текстовую строку.

Листинг 4.3 – Метод парсинга из строки

|  |
| --- |
| bool LibSL::parseFromString(const std::string& content) {  cleanup(); // Очищаем предыдущий парсер  try {  input = std::make\_unique<ANTLRInputStream>(content);  lexer = std::make\_unique<LibSLLexer>(input.get());  tokens =  std::make\_unique<CommonTokenStream>(lexer.get());  parser = std::make\_unique<LibSLParser>(tokens.get());  tokens->fill();  parseTree = parser->file();  return true;  } catch (...) {  cleanup();  return false;  }  } |

Метод parseFromString сначала производит очистку ресурсов. После очистки, метод последовательно объявляет объекты потока данных спецификации, лексер, поток токенов и парсер токенов. Последовательность объявления важна для дальнейшей очистки ресурсов. Данные обрабатываются методами сгенерированного парсера, после работы парсер возвращает результат парсинга и выводит ошибки в случае, если они есть. После завершения, метод сохраняет переменную, содержащую значение лексического дерева.

В листинге 4.4 описывается метод библиотеки, выполняющий парсинг спецификации из файла.

Листинг 4.4 – Метод парсинга из файла

|  |
| --- |
| bool LibSL::parseFromString(const std::string& content) {  cleanup(); // Очищаем предыдущий парсер  try {  input = std::make\_unique<ANTLRInputStream>(content);  lexer = std::make\_unique<LibSLLexer>(input.get());  tokens =  std::make\_unique<CommonTokenStream>(lexer.get());  parser = std::make\_unique<LibSLParser>(tokens.get());  tokens->fill();  parseTree = parser->file();  return true;  } catch (...) {  cleanup();  return false;  }  } |

Метод parseFromString вызывает метод парсинга из строки parseFromString и передает в него данные спецификации, полученные после чтения файла, содержащего спецификацию на языке LibSL. После выполнения процедуры парсинга, метод аналогично сохраняет переменную с сохраненным лексическим деревом.

Далее в листинге 4.5 описывается метод библиотеки, обеспечивающий очистку ресурсов.

Листинг 4.5 – Метод очистки ресурсов

|  |
| --- |
| void LibSL::cleanup() {  if (parseTree) {  delete parseTree;  parseTree = nullptr;  }  parser.reset();  tokens.reset();  lexer.reset();  input.reset();  } |

Метод cleanup сначала очищает ячейку памяти, в котором содержалось лексическое дерево, после чего удаляет соответствующий указатель. После этого, метод очистки последовательно удаляет объекты парсера, потока токенов, лексера и потока данных спецификации.

В листинге 4.6 описывается метод вывода лексического дерева.

Листинг 4.6 – Проверки подключения JVM и генератора парсеров

|  |
| --- |
| std::string LibSL::getParseTree() {  if (!parseTree || !parser) {  return "No parse tree available";  }  return parseTree->toStringTree(&\*parser);  } |

В случае, если переменная parseTree содержит лексическое дерево, метод getParseTree после вызова возвращает структуру дерева в текстовом формате. Данная функциональность опциональна для работы библиотеки парсера с целью проверки корректности файлов спецификации на языке LibSL, однако позволяет в дальнейшем модифицировать и использовать парсер для различных целей. Так, ANTLR4 имеет в своем составе инструменты, позволяющие настраивать и отслеживать правила грамматики используемого языка, то есть языка спецификации LibSL, на основе лексического дерево. Также, данные лексического дерева могут быть использованы в других целях, например, для реализации визуализации лексического дерева. На рисунке 4.4 изображен пример возможной реализации лексического дерева для примера логического выражения на языке LibSL, описанного в листинге 4.7.

Листинг 4.7 – Логическое выражение на языке LibSL, присвоение переменной

|  |
| --- |
| var c: bool = a & a | b & b; |

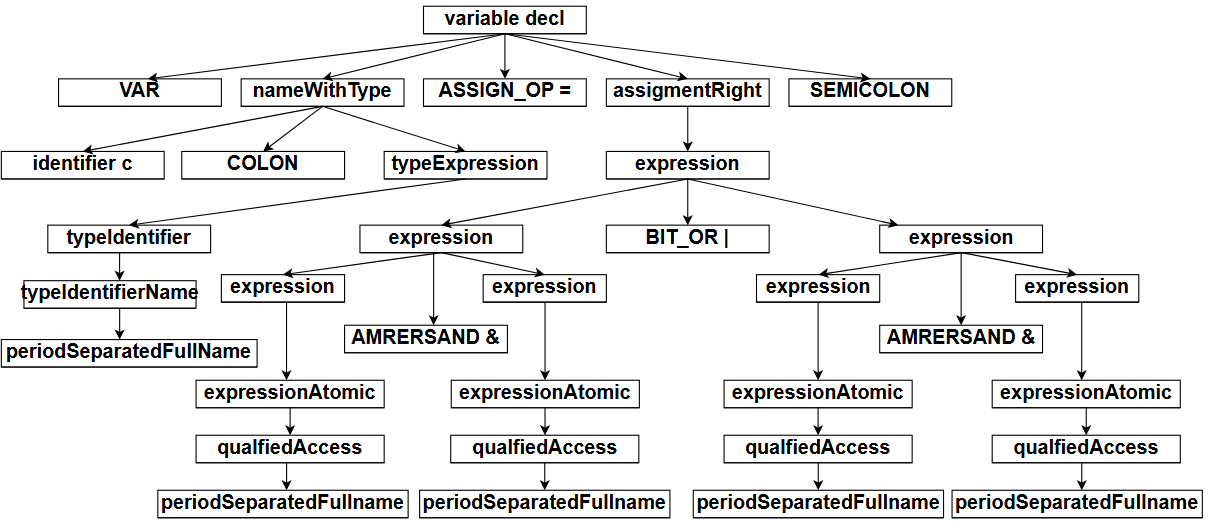


Рисунок 4.3 – Возможная визуализация построенного лексического дерева

## 4.4. Сборка библиотеки парсера

Для сборки библиотеки и ее дальнейшей корректной работы необходимо правильно указать инструкции сборки библиотеки. Основная сложность заключается в относительно большом количестве компонентов библиотеки – сгенерированного парсера, среды выполнения и главного класса библиотеки. Большое количество компонентов требует достаточно тонкой настройки сборки, так как все компоненты должны быть собраны в единый бинарный файл в правильной последовательности.

Перед тем, как внедрить файлы парсера в структуру библиотеки, их необходимо сгенерировать. Для создания файлов парсера должен быть установлен и настроен соответствующий инструмент – генератор парсеров ANTLR4. Несмотря на то, что генератор парсеров позволяет выбирать несколько целевых ЯП для генерации файлов, непосредственно сам генератор парсеров написан на языке Java и для его работы необходимо подключение JVM.

Процедуру генерации файлов парсера можно выполнить один раз, указав через терминал расположение файлов грамматики языка LibSL генератору парсеров ANTLR4, после чего внедрить полученные файлы в структуру библиотеки парсера. Однако, язык спецификации LibSL может потенциально измениться, например, при обновлении грамматики для исправления возможных недочетов или ошибок. В таком случае, будет необходимо выполнить генерацию файлов повторно и заново включить их в структуру библиотеки парсера.

Для предотвращения проблем, связанных с потенциальными изменениями языка спецификации LibSL, необходимо выполнять генерацию файлов парсера перед каждой сборки библиотеки. Для упрощения генерации файлов парсера, процесс их генерации должен быть автоматизирован и описан в инструкциях сборки библиотеки. Также, при сборке библиотеки должен быть настроен генератор парсеров ANTLR4 и к нему должна быть подключена JVM с подходящей версией.

На рисунке 4.4 изображены компоненты, которые задействуются во время сборки библиотеки парсера.

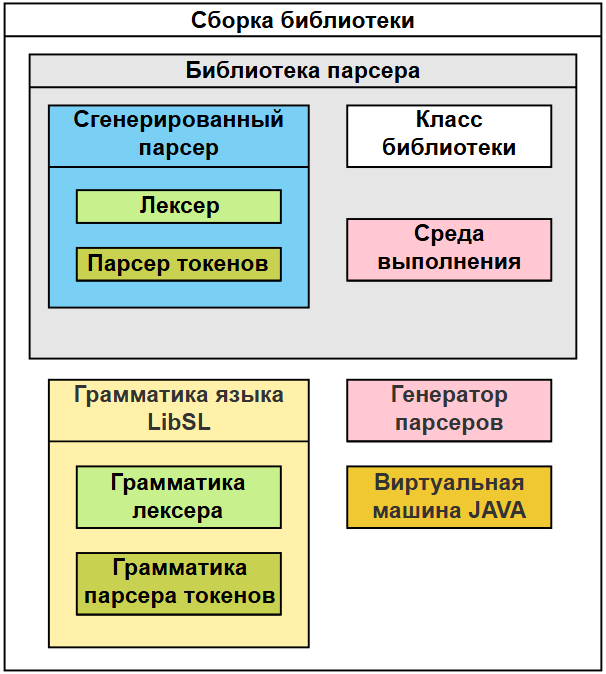


Рисунок 4.4 – Компоненты, задействованные при сборке библиотеки

На рисунке 4.4 изображена внутренняя структура библиотеки, включая сгенерированный парсер, среду выполнения ANTLR4, необходимую для его работы и класс библиотеки. Генерация файлов парсера производится при помощи внешних компонентов - генератора парсеров ANTLR4, который использует файлы грамматики языка LibSL. Для работы генератора парсеров подключается соответствующая JVM.

Процессы сборки библиотеки обеспечиваются при помощи генератора сборочных файлов CMake, для которого указываются инструкции по сборке библиотеки. Подключение необходимых внешних компонентов описывается в листинге 4.8

Листинг 4.8 – Подключение необходимых директорий

|  |
| --- |
| # Настройка ANTLR  set(GRAMMAR\_DIR ${CMAKE\_CURRENT\_SOURCE\_DIR}/grammar)  # Папка с файлами грамматии  set(RUNTIME\_DIR ${CMAKE\_CURRENT\_SOURCE\_DIR}/runtime)  # Папка runtime  set(ANTLR\_JAR ${CMAKE\_CURRENT\_SOURCE\_DIR}/../libs/antlr-4.13.2-complete.jar)  # Настройка JAVA, необходимая для работы ANTLR\_JAR set(JAVA\_EXECUTABLE "C:/Program Files/Java/jdk17/bin/java.exe") |

В листинге 4.8 производится подключение директорий, содержащих файлы грамматики, среду выполнения ANTLR4, генератора парсеров ATNLR4 и расположение установленной JVM, необходимой для работы генератора парсеров.

Далее, выполняются проверки подключения установленной JVM и генератора парсеров, так как в случае, если они не будут по какой-либо причине подключены или подключены с ошибкой, то сборка библиотеки может быть выполнена без включения указанных компонентов и библиотека парсера не будет работать корректно. Соответственно, в случае если такая ошибка произошла, то сборка библиотеки парсера будет преждевременно остановлена. Проверка подключения необходимых компонентов описывается в листинге 4.9.

Листинг 4.9 – Проверки подключения JVM и генератора парсеров

|  |
| --- |
| # Проверки  if(NOT EXISTS "${JAVA\_EXECUTABLE}")  message(FATAL\_ERROR "Java not found at:${JAVA\_EXECUTABLE}")  endif()  if(NOT EXISTS "${ANTLR\_JAR}")  message(FATAL\_ERROR "ANTLR JAR not found at: ${ANTLR\_JAR}")  endif() |

Далее необходимо указать и подключить целевую директорию, в которой будут размещаться сгенерированные файлы парсера и перечислить файлы, которые будут получены после работы генератора парсеров. Создание директории также автоматизировано для предотвращения возможных ошибок при сборке. Соответствующие инструкции описываются в листинге 4.10.

Листинг 4.10 – Подготовка к генерации файлов парсера

|  |
| --- |
| # Создание папки для генерации  file(MAKE\_DIRECTORY "${GENERATED\_DIR}")  set(GENERATED\_DIR ${CMAKE\_CURRENT\_BINARY\_DIR}/generated)  # Список ожидаемых файлов  set(GENERATED\_SOURCES  ${GENERATED\_DIR}/LibSLLexer.cpp  ${GENERATED\_DIR}/LibSLParser.cpp  ${GENERATED\_DIR}/LibSLParserBaseVisitor.cpp  ${GENERATED\_DIR}/LibSLParserVisitor.cpp  ${GENERATED\_DIR}/LibSLParserBaseListener.cpp  ${GENERATED\_DIR}/LibSLParserListener.cpp  ${GENERATED\_DIR}/LibSLLexer.h  ${GENERATED\_DIR}/LibSLParser.h  ) |

Ключевыми файлами парсера, которые генерируются в процессе сборки, являются файлы с названиями LibSLLexer и LibSLParser. Именно данные файлы обеспечивают логику работы парсера языка LibSL. Остальные файлы опциональны для выполнения задачи по разработке библиотеки парсера. Они предназначены для настройки грамматики языка LibSL и могут быть использованы при необходимости в дальнейшем.

После того, как все необходимые внешние компоненты подготовлены и подключены выполняется процесс генерации файлов парсера. Соответствующие инструкции описываются в листинге 4.11.

Листинг 4.11 – Подготовка к генерации файлов парсера

|  |
| --- |
| # Указываем исходные грамматики  set(GRAMMAR\_FILES  ${GRAMMAR\_DIR}/LibSLLexer.g4  ${GRAMMAR\_DIR}/LibSLParser.g4  )  # Генерация ANTLR  add\_custom\_command(  OUTPUT ${GENERATED\_SOURCES}  COMMAND ${JAVA\_EXECUTABLE} -jar ${ANTLR\_JAR}  -Werror  -Dlanguage=Cpp  -listener -visitor  -o "${GENERATED\_DIR}"  ${GRAMMAR\_FILES}  DEPENDS ${GRAMMAR\_FILES}  COMMENT "Generating ANTLR parser files"  VERBATIM  )  # Цель генерации  add\_custom\_target(GenerateParser DEPENDS ${GENERATED\_SOURCES}) |

Файлы парсера генерируются на основе файлов с описаниями правил грамматики LibSLexer.g4 и LibSLParser.g4. Далее выполняется команда генерации файлов, которую выполняет генератор парсеров ANTLR4. В команде указываются подключенные JVM и исполняемый файл генератора парсеров, а также расположение сгенерированных файлов.

После того файлы парсера сгенерированы, необходимо произвести их линковку со средой выполнения, обеспечивающей их работу, и главным классом библиотеки. Полученный в результате бинарный файл и будет представлять собой готовую библиотеку парсера. Соответствующие инструкции описаны в листинге 4.12.

Листинг 4.12 – Сборка компонентов в библиотеку

|  |
| --- |
| # ANTLR runtime  add\_subdirectory(${RUNTIME\_DIR})  include\_directories(${RUNTIME\_DIR}/runtime/src)  # Основные файлы  file(GLOB LIBSL\_SRC CONFIGURE\_DEPENDS  ${CMAKE\_CURRENT\_SOURCE\_DIR}/LibSL.cpp  )  add\_library(libsl-parser STATIC ${LIBSL\_SRC} ${GENERATED\_SOURCES})  add\_dependencies(libsl-parser GenerateParser)  # Подключаем директории  target\_include\_directories(libsl-parser PUBLIC  ${CMAKE\_CURRENT\_SOURCE\_DIR}/include  ${GENERATED\_DIR}  )  target\_link\_libraries(libsl-parser PUBLIC antlr4\_static)  target\_compile\_features(libsl-parser PRIVATE cxx\_std\_20) |

Полученную в процессе сборку библиотеку парсера языка LibSL можно применять для решения различных задач, в том числе для проверок корректности используемых файлов спецификации.

## 4.6. Подключение библиотеки

Для корректного использования библиотеки парсера необходимо правильно подключить ее к программе. Пример программы, которая использует библиотеку парсера приведен в листинге 4.13.

Листинг 4.13 – Программа, использующая библиотеку парсера

|  |
| --- |
| #include "LibSL.h"  #include <iostream>  #include <fstream> |

Продолжение листинга 4.13

|  |
| --- |
| int main() {  const std::string filePath ="project/examples/example.lsl";  if (LibSL::parseFromFile(filePath)) {  std::cout << "File parsed successfully!" << std::endl;  std::cout << "\n=== Parse Tree ===\n";  std::cout << LibSL::getParseTree() << std::endl;  return 0;  } else {  std::cerr << "Failed to parse file!" << std::endl;  return 1;  }  } |

Программа, описанная в листинге 4.13, открывает файл спецификации по указанному пути, выполняет процедуру парсинга данных спецификации, после чего выдает результат парсинга и выводит лексическое дерево в текстовом формате. В листинге 4.14 описываются инструкции сборки, которые собирают исходный код программы и библиотеку парсера в единый исполняемый бинарный файл.

Листинг 4.14 – Инструкции сборки программы с библиотекой парсера

|  |
| --- |
| project(executable)  set(EXECUTABLE\_SOURCE\_DIR ${CMAKE\_SOURCE\_DIR}/executable)  add\_executable(executable main.cpp)  # Подключаем библиотеку  target\_link\_libraries(executable PRIVATE libsl-parser) |

# тестирование

Для подтверждения успеха разработки библиотеки парсера, необходимо провести тестирование, чтобы продемонстрировать, что разработанная библиотека парсера соответствует поставленным требованиям и выполняет поставленные задачи. В соответствии с требованиями были проведены положительные и негативные тесты.

Тестирование проводилось с использованием файлах спецификации на языке LibSL. В качестве первого этапа тестирования использовались файлы спецификации, которые использовались для регрессионного тестирования библиотеки парсера на ЯП Kotlin [1][2]. Репозиторий с тестовыми примерами: <https://github.com/vpa-research/libsl-parser/tree/main/src/test>. Были проведены положительные тесты, в ходе которых было установлено, что результаты работы библиотеки парсера на ЯП Kotlin идентичны работе библиотеки на ЯП С++. Далее были проведены негативные тесты – было выполнено несколько итераций, и парсеры получали на вход некорректно описанную грамматику. Было установлено, что вывод грамматических ошибок обоих парсеров также идентичен.

Для второго этапа тестирования были использованы файлы спецификации, описывающие поведение стандартной библиотеки, написанной на Java. Спецификация довольно обширна и содержится больше, чем в сотне файлов. Репозиторий с файлами спецификации: <https://github.com/vpa-research/jsl-spec/tree/main/spec>. В ходе положительного тестирования было установлено, что разработанная библиотека парсера успешно выполняет парсинг файлов спецификации и выдает положительный ответ.

Таким образом было подтверждено, что разработанная библиотека парсера соответствует заданным требованиям. Также возможны более глубокие подходы к тестированию, с анализом уязвимых мест грамматики языка LibSL, более широкого применения негативных тестов и т.д.

# экономическое обоснование

## 6.1. Концепция

Целью выпускной квалификационной работы является разработка библиотеки парсера для языка спецификации библиотек на языке С++. В рамках ВКР были реализованы программная реализация прототипа парсера и библиотечная реализация парсера, а также проведено их тестирование.

Для разработки использовались платные коммерческие среды разработки – кроссплатформенная IDE CLion для работы с С++ и IDE IntelliJ IDEA для работы на Java и Kotlin. Обе среды разработки поставляются компанией JetBrains.

## 6.2. Расходы на оплату труда

Для расчета себестоимости проектирования и разработки продукта необходимо рассчитать затраты на оплату труда. Для этого необходимо сформировать план проводимых работ Работу предполагается выполнить силами двух разработчиков и одного специалиста QA. Для расчета заработной платы были взяты данные с сайта поиска работы hh.ru. Было установлено, что средняя заработная плата, указываемая в вакансиях «Разработчик C/С++» приблизительно составляет 120 000 (Сто двадцать тысяч) рублей [8][9][10]. Средняя заработная плата специалиста QA составляет 150 000 (Сто пятьдесят тысяч) рублей [11][12][13]. На основании полученных данных необходимо рассчитать дневные ставки разработчиков на С++ и специалиста QA по формулам 6.1 и 6.2 соответственно.

Где – это дневная заработная плата сотрудника-разработчика, а – дневная заработная плата сотрудника QA-специалиста.

Для расчета затрат на этапах проектирования и разработки библиотеки парсера необходимо сформировать план необходимых работ. При помощи сформированного плана работ нужно определить трудоемкость работ и рассчитать стоимость их выполнения. Планы необходимых работ для разработчика и для QA-специалиста представлены в таблице 6.1 и таблице 6.2 соответственно.

Таблица 6.1 – План необходимых работ разработчиков.

| № | Этапы и содержание выполняемых работ | Трудоемкость, чел/день | Ставка, руб./день |
| --- | --- | --- | --- |
| 1 | Анализ предметной области | 5 | 5714,28 |
| 2 | Формирование технического задания | 3 | 5714,28 |
| 3 | Проектирование библиотеки парсера | 20 | 5714,28 |
| 4 | Разработка программного прототипа | 5 | 5714,28 |
| 5 | Разработка библиотеки парсера | 10 | 5714,28 |
| 6 | Оформление пояснительной записки | 10 | 5714,28 |

Таблица 6.2 – План необходимых работ QA-специалиста.

| № | Этапы и содержание выполняемых работ | Трудоемкость, чел/день | Ставка, руб./день |
| --- | --- | --- | --- |
| 1 | Тестирование прототипа | 5 | 7142,85 |
| 2 | Тестирование библиотеки | 10 | 7142,85 |

Данные о трудоемкости выполнения работ указаны в человеко-днях и приведены в таблицах 6.1 и 6.2 соответственно. Затраты на основную заработную плату исполнителей указаны в таблице 6.3.

Таблица 6.3 – Суммарные расходы на основную заработную плату исполнителей

|  |  |  |  |
| --- | --- | --- | --- |
| Исполнитель | Оплата, руб./день | Количество дней | Основная оплата за выполнение проекта, руб |
| Разработчик | 5714,28 | 53 | 302857.14 |
| QA-специалист | 7142,85 | 15 | 107142,84 |
| Итого: | | | 410000 |

Далее необходимо определить затраты на дополнительную заработную плату по формуле 6.3.

где - расходы на дополнительную заработную плату исполнителей (руб.), - расходы на основную заработную плату исполнителей (руб.), - норматив дополнительной заработной плат, в рамках данной работы принимается равным 8%.

Расчет расходов на дополнительную заработную плату в соответствии с формулой 6.3 и суммарной величины заработной платы исполнителей представлен в таблице 6.4.

Таблица 6.4 – Суммарные отчисления на дополнительную заработную плату исполнителей

|  |  |  |  |
| --- | --- | --- | --- |
| Исполнитель | Оплата, руб./день | Количество дней | Дополнительная оплата за выполнение проекта, руб |
| Разработчик | 457,14 | 53 | 24228,57 |
| QA-специалист | 571,42 | 15 | 8571,42 |
| Итого: | | | 32800 |

## 6.3. Отчисления на социальные нужды

Для расчета себестоимости проектирования и разработки продукта также необходимо посчитать отчисления на социальные нужды. Отчисления во внебюджетные фонды на обязательное социальное, медицинское и пенсионные страхования необходимо рассчитать исходя из величины основной и дополнительной заработной платы исполнителей по формуле 6.4.

где – отчисления на социальные нужды с заработной платы (руб.), – затраты на основную заработную плату исполнителей (руб.), – затраты на дополнительную заработную плату исполнителей (руб.), – норматив отчислений на страховые взносы на обязательное социальное, пенсионное и медицинское страхование (%), в рамках данной работы принимается равным 30 %.

Затраты социальных отчислений для разработчика рассчитываются по формуле 6.5

Затраты социальных отчислений для QA-специалиста рассчитываются по формуле 6.6

Результаты расчёта отчислений на социальные нужды для разработчика и QA-специалиста приведены в табл. 6.5.

Таблица 6.5 – Расходы на социальные нужды

|  |  |  |  |
| --- | --- | --- | --- |
| Исполнитель | Оплата, руб./день | Количество дней | Отчисления на социальные нужды, руб |
| Разработчик | 1851,42 | 53 | 98125,71 |
| QA-специалист | 2777,14 | 15 | 41657,14 |
| Итого: | | | 139782,85 |

## 6.4. Материалы

Для расчета себестоимости проектирования и разработки продукта также необходимо посчитать расходы на материалы. Также необходимо учесть транспортно-заготовительные расходы. Расчеты необходимо произвести по формуле 6.7.

где – затраты на сырье и материалы (руб.), L – индекс вида сырья, – норма расхода l-го материала (руб./ед.), – норма транспортно-заготовительных расходов, в рамках данной работы принимается равной 10%.

Список материалов и расчет расходов на них в соответствии с формулой 6.7 представлен в таблице 6.6.

Таблица 6.6 – Расчет расходов на материалы

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Изделие | Тип | Норма расходов на изделие (ед.) | Цена за единицу согласно данным  OZON (руб./шт.) | Сумма на изделие (руб.) |
| Авторучки | шариковые автоматические синие R-301 Classic, 8 штук | 1 | 294 [14] | 294 |
| Бумага офисная "SvetoCopy" | 500 листов, А4. | 1 | 347 [15] | 347 |
| Итого | | | | 641 |
| Транспортно-заготовительные расходы (10%) | | | | 64,1 |
| Итого с учетом транспортно-заготовительных расходов | | | | 705,1 |

## 6.5. Амортизационные отчисления

Для расчета себестоимости проектирования и разработки продукта необходимо рассчитать амортизационные отчисления. Амортизационные отчисления по основному средству за год определяются по формуле 6.8.

где – амортизационные отчисления за год по i-му основному средству (руб.), – первоначальная стоимость i-го основного средства (руб.), – годовая норма амортизации i-го основного средства (%).

Расчет расходов на годовые амортизационные отчисления по используемому оборудованию представлен в таблице 6.7. Срок полезного использования определен на основании Постановления Правительства РФ от 01.01.2002 N 1 (ред. от 28.04.2018) «О Классификации основных средств, включаемых в амортизационные группы» [16]. Норма амортизации рассчитана по формуле 6.9:

Таблица 6.7 – Расчет амортизационных отчислений за год

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Устройство | Первоначальная стоимость (руб.) | Срок полезного использования (год) | Годовая норма амортизации (%) | Амортизационные отчисления за год (руб.) |
| Ноутбук HP ProBook 650 G8 | 288 703 [17] | 10 | 10 | 28870,3 |
| Принтер струйный Canon PIXMA G1430 | 13 499 [18] | 10 | 10 | 1349,9 |

Амортизационные отчисления по i-му основному средству, используемому разработчиком при работе над ВКР, определяется по формуле 6.10.

где – амортизационные отчисления по i-му основному средству, используемому студентом в работе над ВКР (руб.); – амортизационные отчисления за год по i-му основному средству (руб.); – время, в течение которого разработчик использует i-ое основное средство (мес.).

Расчет расходов на амортизационные отчисления по оборудованию, используемому при работе над ВКР, представлен в таблице 6.8.

Таблица 6.8 – Расчет амортизационных отчислений

|  |  |  |  |
| --- | --- | --- | --- |
| Устройство | Амортизационные отчисления за год (руб.) | Время выполнения ВКР (мес.) | Амортизационные отчисления (руб.) |
| Ноутбук HP ProBook 650 G8 | 28870,3 | 5 | 12029,29 |
| Принтер струйный Canon PIXMA G1430 | 1349,9 | 5 | 562,45 |
| Итого: | | | 12491,75 |

## 6.6 Расчет итоговой себестоимости продукта

Для расчета итоговой себестоимости продукта были посчитаны все статьи расходов – расходы на оплату труда, отчисления на социальные нужды, затраты на материалы и амортизационные отчисления. Расчет итоговой себестоимости продукта указан в таблице 6.9.

Таблица 6.9 – Расчет итоговой себестоимости продукта

|  |  |  |
| --- | --- | --- |
| Наименование статьи | Сумма (руб.) | Структура себестоимости, (%) |
| Расходы на оплату труда | 410000 | 68,81 |
| Отчисления на доп. заработную плату | 32800 | 5.50 |
| Отчисления на социальные нужды | 139782,85 | 23,46 |
| Материалы | 705,1 | 0,11 |
| Амортизационные отчисления | 12491,75 | 2,14 |
| Итого | 595779.7 | 100 |

## 6.7 Вывод

В экономическом разделе выпускной дипломной работы были подсчитаны статьи расходов на ее выполнение. Были учтены затраты на оплату труда, отчисления на социальные нужды, затраты на материалы и амортизационные отчисления. Было установлено, что общая себестоимость разработки продукта в ходе выполнения работы составила 595779,7 рублей (пятьсот девяносто пять тысяч семьсот семьдесят девять рублей и 70 копеек).

заключение

В результате выполнения дипломной работы была реализована библиотека парсера языка спецификации LibSl. На этапе проектирования была составлена наиболее оптимальная структура библиотеки парсера, которая в дальнейшем использовалась на этапе разработки библиотеки. Библиотека парсера соответствует заявленным принципам и требованиям и является актуальным решением для использования языка LibSl и проверки грамматики файлов соответствующей спецификации.

Также в ходе разработки был реализован такой важный аспект, как использование грамматики на основе ANTLR4, что позволяет получать детальные отчеты об ошибках в файлах спецификации с четким указанием того, какие элементы языка спецификации описаны некорректно.

Структура библиотеки парсера позволяет применять ее статический и динамический варианты реализации на ОС Windows. Кроме того, библиотека парсера может быть модернизирована доступными средствами для использования на дистрибутивах ОС Linux. Соответствие библиотеки поставленным требованиям осуществляется при помощи функциональных тестов, проверяющих вывод парсера на заранее заготовленные тестовые файлы с грамматикой.

Результат дипломной работы в виде библиотеки парсера расширяет доступность применения языка спецификации LibSl, что позволяет упростить процессы составления и использования спецификации и автоматизировать процессы, связанные с интеграцией библиотек в программные продукты.

список использованных источников

1. «Высшая школа экономики» (НИУ ВШЭ) [Электронный ресурс] URL: https://nnov.hse.ru/ba/se/students/diplomas/925717980 (Дата обращения: 18.03.2025)

2. Ицыксон В. М. LibSL-язык спецификации компонентов программного обеспечения //Программная инженерия. – 2018. – Т. 9. – №. 5. – С. 209.

3. Aleksyuk A. O., Itsykson V. M. Semantics-driven migration of Java programs: a practical application //Automatic Control and Computer Sciences. – 2018. – Т. 52. – С. 581-588.

4. Feofilaktov V., Itsykson V. SPIDER: Specification-Based Integration Defect Revealer //International Conference on Software Testing, Machine Learning and Complex Process Analysis. – Cham : Springer Nature Switzerland, 2021. – С. 107-119.

5. Hanson D. R. Compact recursive‐descent parsing of expressions //Software: Practice and Experience. – 1985. – Т. 15. – №. 12. – С. 1205-1212.

6. Parr T. The definitive ANTLR 4 reference. – 2013.

7. Mecenero L. et al. Lightweight Parsing and Slicing for Bug Identification in C //Proceedings of the 17th International Conference on Availability, Reliability and Security. – 2022. – С. 1-10.

8. Сайт вакансий hh.ru [Электронный ресурс] URL:<https://spb.hh.ru/vacancy/119899379?query=C%2B%2B+%D1%82%D0%B5%D1%81%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5&hhtmFrom=vacancy_search_list>

(дата посещения: 12.05.2025)

9. Сайт вакансий hh.ru [Электронный ресурс] URL:  
<https://spb.hh.ru/vacancy/120483733?query=%D0%A1%2B%2B&hhtmFrom=vacancy_search_list>  
(дата посещения: 12.05.2025)

10. Сайт вакансий hh.ru [Электронный ресурс] URL:  
<https://spb.hh.ru/vacancy/120601559?query=%D0%A1%2B%2B&hhtmFrom=vacancy_search_list>

(дата посещения: 12.05.2025)

11. Сайт вакансий hh.ru [Электронный ресурс] URL:  
<https://spb.hh.ru/vacancy/120287789?query=QA&hhtmFrom=vacancy_search_list>

(дата посещения: 12.05.2025)

12. Сайт вакансий hh.ru [Электронный ресурс] URL:  
<https://spb.hh.ru/vacancy/120643219?query=QA&hhtmFrom=vacancy_search_list>

(дата посещения: 12.05.2025)

13. Сайт вакансий hh.ru [Электронный ресурс] URL:  
<https://spb.hh.ru/vacancy/119930883?query=QA&hhtmFrom=vacancy_search_list>

(дата посещения: 12.05.2025)

14. Интернет-магазин OZON [Электронный ресурс] URL:  
<https://www.ozon.ru/product/ruchki-sharikovye-avtomaticheskie-sinie-r-301-classic-8-shtuk-530311067/?at=w0tgEgXZMsBDo00NiZ8wBw1fwp1lLmuV4gP6YT2Mj0qP>

(дата посещения: 12.05.2025)

15. Интернет-магазин OZON [Электронный ресурс] URL:  
<https://www.ozon.ru/product/bumaga-ofisnaya-svetocopy-500-listov-a4-000877-1619822268/?at=RltyDyKRWIRkxZJBtVMLJPjI8P1OGWHVMo7y5CAA1ZRJ>

(дата посещения: 12.05.2025)

16. О классификации основных средств, включаемых в амортизационные группы: постановление Правительства Рос. Федерации от 01.01.2002 №1 (ред. от 28.04.2018) // Собрание законодательства Российской Федерации.

17. Интернет-магазин Яндекс маркет [Электронный ресурс] URL: <https://market.yandex.ru/product--15-6-noutbuk-hp-probook-650-g8-3s8n9ea-serebristyi-1920x1080-ips-intel-core-i5-1135g7-iadra-4-x-2-4-ggts-8-gb-ssd-256-gb-intel-iris-xe-graphics-windows-10-pro/1152623524?sku=4375387515&uniqueId=895648&do-waremd5=OyLlJOcJO80_kpYX2CieyQ&cpc=b1LImdgdY6vW5EwKJDlWyaSWmGGk9srVP9Slsx2s794WH0V7ryndYCqFOOSKAbCoAiTziwrPx7wKlljhvRMABaydd30Yc2_NgjeWAbhon24NN9Btn8fBtDF-EeU-vSuTPIIVzE7pnsWgI03m7KAoTm8YIMrPHIygjLeZlgOVoQnq0fYm0dcsOo37lJ5ygx3_2G7U5om7j5yv-1g_lVodi_BiBBIH-UaZ0AmfqEpigdZknCInn5IGzUKPyFsQz3q_5rFTQZy2FKp5C55tewg450fXxXjrvHISlf3E_S5Cgst0z2MtmbbE7C4uzO8t4e_3XWruTsbJOUwvFvc5x2NnwI2S3yMx40LSaFBY7Ys63mNddznfgg3iyQl6ekW2fLRiT2UqyoINXoeHUxSefDWWcvbX6_XDP9gC10vb_q-BvUC3YO69IHic6wTMm5_BEQe4M6584KwN5unLTT2pgfhzbww8KEbBHu3xWY3nX7Y9nGIFSttabEFk3VgbqNOp82t99CUEZgiXHvM%2C>

(дата посещения: 12.05.2025)

18. Интернет-магазин DNS [Электронный ресурс] URL:

<https://www.dns-shop.ru/product/3c56fb23ed77d9cb/printer-strujnyj-canon-pixma-g1430/?utm_referrer=https%3A%2F%2Fwww.dns-shop.ru%2Fcatalog%2Frecipe%2F79247710b7c74278%2Fstrujnye-printery-canon%2F%3Futm_medium%3Dorganic%26utm_source%3Dgoogle%26utm_referrer%3Dhttps%253A%252F%252Fwww.google.com%252F>

(дата посещения: 12.05.2025)