МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное автономное образовательное учреждение  
 высшего образования

**«Дальневосточный федеральный университет»**

**ИНСТИТУТ МАТЕМАТИКИ И КОМПЬЮТЕРНЫХ ТЕХНОЛОГИЙ (ШКОЛА)**

**Департамент программной инженерии и искусственного интеллекта**

РАЗРАБОТКА ТРАНСЛЯТОРА, ПЕРЕВОДЯЩЕГО ПОДМНОЖЕСТВО ЯЗЫКА SHELL В ЭКВИВАЛЕНТНОЕ ПОДМНОЖЕСТВО ЯЗЫКА JAVASCRIPT

**ОТЧЁТ ПО ЛАБОРАТОРНОЙ РАБОТЕ**

по дисциплине «Теория языков программирования и компиляторы»

по образовательной программе подготовки бакалавров по направлению 09.03.04 Программная инженерия

Выполнили студенты

гр. Б9120-09.03.04прогин

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Колесник А.С.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Нельбасов Д.М.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Перепечин В.В.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Алексеев В.М.

Руководитель:

ассистент департамента ПИиИИ

Глазырин М.А.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_ «\_\_\_» \_\_\_\_\_\_\_\_\_\_ 2024 г.

(подпись)

г. Владивосток

2024 г

# Оглавление

[Оглавление 2](#_Toc153795123)

[Введение 3](#_Toc153795124)

[1 Неформальная постановка задачи 6](#_Toc153795125)

[2 Синтаксис входного языка 6](#_Toc153795126)

[3 Контекстные условия языка Shell 7](#_Toc153795127)

[4 Контекстные условия языка JavaScript 9](#_Toc153795128)

[5 Таблица соответствия языков 10](#_Toc153795129)

[6 Проект лексического анализатора 11](#_Toc153795130)

[6.1 Модель данных лексического анализатора: 11](#_Toc153795131)

[6.2 Таблица ключевых слов: 13](#_Toc153795132)

[6.3 Таблица зарезервированных имен: 14](#_Toc153795133)

[6.4 Типы лексем: 15](#_Toc153795134)

[6.5 Типы ошибок: 15](#_Toc153795135)

[6.6 Конечный автомат лексического анализатора: 16](#_Toc153795136)

[7 Проект синтаксического анализатора 16](#_Toc153795137)

[7.1 Модель данных: 17](#_Toc153795138)

[7.2 Описание алгоритма рекурсивного спуска семантического анализа: 17](#_Toc153795139)

[7.3 Пример дерева разбора 19](#_Toc153795140)

[7.4 Типы ошибок: 19](#_Toc153795141)

[8 Проект семантического анализатора 22](#_Toc153795142)

[8.1 Методы проверки нарушений контекстных условий 24](#_Toc153795143)

[8.2 Типы ошибок 24](#_Toc153795144)

[9 Проект генератора кода 25](#_Toc153795145)

[9.1 Метод кодогенерации 25](#_Toc153795146)

[9.2 Пример кодогенерации 25](#_Toc153795147)

[9.3 Оптимизация и методы оптимизации 26](#_Toc153795148)

[10 Тестирование 27](#_Toc153795149)

[10.1 Лексический анализатор 28](#_Toc153795150)

[10.2 Синтаксический анализатор 31](#_Toc153795151)

[10.3 Семантический анализатор 32](#_Toc153795152)

[10.4 Генератор кода 33](#_Toc153795153)

[Заключение 35](#_Toc153795154)

# Введение

В мире программирования, транслятор играет ключевую роль, обеспечивая перевод кода из одного языка программирования в другой. Этот процесс не только улучшает совместимость между различными технологиями, но и расширяет возможности разработчиков при работе с разнообразными инструментами и платформами.

Язык программирования — это средство, позволяющее разработчикам давать инструкции компьютеру. Каждый язык имеет свои уникальные особенности и предназначен для решения определенных задач. В нашем проекте, мы сосредоточимся на двух языках: Shell и JavaScript. Shell, который мы выбрали в качестве входного языка, известен своей мощью в автоматизации задач и управлении системными процессами в UNIX-подобных операционных системах. JavaScript, выбранный в качестве выходного языка, является одним из самых популярных языков для веб-разработки, обладая широкими возможностями для создания интерактивных веб-приложений.

Целью нашего проекта является создание транслятора, который будет переводить скрипты, написанные на Shell, в код JavaScript. Это позволит разработчикам легко интегрировать сценарии Shell в веб-приложения и расширить их функционал, используя возможности JavaScript. Такой подход открывает новые горизонты в разработке, позволяя объединить мощь Shell для работы с системой и гибкость JavaScript для создания богатых веб-интерфейсов.

**Цель лабораторной работы:** разработать проект транслятора, который преобразует программу, содержащую подмножество языка Shell, в программу, содержащую подмножество языка Javascript генерируя эквивалентный исходный код.

**Задачи:**

1. Описать грамматику подмножества входного языка;
2. Описать контекстные условия входного и выходного языков;
3. Описать соответствие конструкций входного и выходного языков;
4. Разработать проект лексического анализатора;
5. Разработать проект синтаксического анализатора;
6. Разработать проект семантического анализатора;
7. Разработать проект генератора кода выходного языка;
8. Реализовать транслятор входного языка в выходной;
9. Выполнить тестирование разработанного транслятора.

**Определения:**

Транслятор - это программное обеспечение, которое принимает входные данные на одном языке и производит эквивалентные данные на другом языке. Он выполняет преобразование программы из одного языка программирования в другой, сохраняя функциональность и семантику исходного кода.

Язык программирования - это формальный набор символов, правил и синтаксиса, используемых для написания программ компьютера. Язык программирования определяет набор инструкций, которые компьютер может выполнить для решения определенной задачи.

В данной лабораторной работе мы выбрали язык Shell в качестве входного языка и язык JavaScript в качестве выходного языка. Язык Shell является командным интерпретатором в Unix-подобных операционных системах, который предоставляет пользователю возможность взаимодействия с операционной системой через командную строку. Язык JavaScript, в свою очередь, широко используется для разработки веб-приложений и обладает богатыми возможностями в области веб-разработки.

Основная цель нашего проекта заключается в автоматизации и упрощении процесса перевода программ, написанных на языке Shell, в программы на языке JavaScript. Это позволит разработчикам использовать уже существующий код на Shell для создания веб-приложений на JavaScript, экономя время и ресурсы.

Далее в отчете будут представлены подробности о процессе разработки транслятора, используемых алгоритмах и методах, а также приведены примеры работы транслятора на различных входных программах на языке Shell.

# 1 Неформальная постановка задачи

Целью данного проекта является разработка транслятора, способного переводить подмножество языка Shell в эквивалентное подмножество языка JavaScript. Мы выбрали следующее подмножество языка Shell, которое будет поддерживаться транслятором:

1. Типы данных - в нашем подмножестве поддерживаются целочисленные (integer), числа с плавающей точкой (float), строки (string) и логический (bool) типы данных.

2. Операторы - включаются арифметические операторы (например, сложение, вычитание), логические операторы (например, логическое И), а также другие необходимые операторы.

3. Операторы циклов - в нашем подмножестве поддерживаются операторы циклов while, for которые позволяют выполнять повторяющиеся действия пока это необходимо.

4. Операторы ветвления - поддерживается оператор ветвления if, который позволяет выполнять определенные действия в зависимости от условия.

5. Стандартные функции - поддерживаются стандартные функции языка Shell, такие как функция вывода на экран (print) и другие необходимые стандартные функции.

При разработке транслятора мы уделяем особое внимание сохранению функциональности и семантики исходного кода, чтобы перевод был точным и корректным.

# 2 Синтаксис входного языка

Входной язык Shell, который мы выбрали для разработки транслятора, имеет свой уникальный синтаксис. Ниже представлены основные обозначения, используемые в описании грамматики входного языка:

* "\n" - обозначает символ переноса строки.
* "\t" - обозначает символ табуляции.

Грамматика входного языка может быть описана следующим образом:

* <PROGRAM> ::= <INSTRUCT> | <INSTRUCTIONS>
* <INSTRUCTIONS> ::= <INSTRUCT> <INSTRUCTIONS>

| <INSTRUCT>

* <INSTRUCT> ::= <STATEMENT>

| <LOOP>

| <CONDITION>

* <STATEMENT> ::= <VARIABLE> <OPERATOR> <VALUE>
* <VARIABLE> ::= Имя переменной
* <OPERATOR> ::= Арифметический оператор или оператор присваивания
* <VALUE> ::= Значение переменной или литерал
* <LOOP> ::= "while" <CONDITION> <INSTRUCT>
* <CONDITION> ::= "if" <EXPRESSION> "then" <INSTRUCT>
* <EXPRESSION> ::= Логическое выражение
* Арифметические операторы: сложение (+), вычитание (-), умножение (\*), деление (/) и т. д.
* Логические операторы: логическое И (&&), логическое ИЛИ (||), логическое НЕ (!) и т. д.
* Операторы присваивания: "=" (присвоение значения переменной).

Грамматика представляет собой формальное описание структуры языка и определяет правила комбинирования токенов (лексем) для создания корректных выражений и конструкций в языке. Эта грамматика будет использоваться в процессе разработки транслятора для анализа и трансформации программ на языке Shell в эквивалентные программы на языке JavaScript.

# 3 Контекстные условия языка Shell

Язык Shell, как и любой другой язык программирования, имеет свои контекстные условия, которые ограничивают использование определенных конструкций или определяют особенности их использования. Ниже приведены некоторые контекстные условия, характерные для языка Shell:

1. Права доступа - в языке Shell некоторые команды и операции могут быть ограничены правами доступа пользователя. Например, выполнение определенных команд может требовать прав администратора или определенного уровня разрешений.

2. Переменные окружения - язык Shell использует переменные окружения для хранения значений, которые могут быть использованы в командах и скриптах. Контекстные условия могут определять доступные переменные окружения и их значения.

3. Синтаксические ограничения - язык Shell имеет свой синтаксис и правила написания команд. Контекстные условия могут определять, какие конструкции считаются правильными или неправильными, и какие символы или ключевые слова допустимы в определенных контекстах.

4. Встроенные команды и функции - язык Shell предоставляет набор встроенных команд и функций, которые могут использоваться для выполнения различных операций. Контекстные условия могут определять доступные встроенные команды и их использование в определенных ситуациях.

5. Взаимодействие с операционной системой - язык Shell предоставляет возможность взаимодействия с операционной системой, вызывая различные системные команды и программы. Контекстные условия могут ограничивать или управлять доступом к определенным системным ресурсам или командам.

Учет контекстных условий в разработке транслятора является важным аспектом, поскольку позволяет гарантировать совместимость и корректность перевода программ на языке Shell в эквивалентные программы на языке JavaScript. При разработке транслятора необходимо учитывать контекстные условия и обрабатывать их соответствующим образом, чтобы получить правильные результаты в выходном коде.

# 4 Контекстные условия языка JavaScript

Язык JavaScript также имеет свои контекстные условия, которые ограничивают использование определенных конструкций или определяют особенности их использования. Ниже приведены некоторые контекстные условия, характерные для языка JavaScript:

1. Область видимости переменных - JavaScript имеет концепцию области видимости переменных, таких как глобальная область видимости и область видимости функций. Контекстные условия определяют доступность переменных в определенных областях видимости и область видимости переменных внутри функций.

2. Строгий режим (Strict mode) - JavaScript поддерживает строгий режим, который активируется с помощью специальной директивы "use strict". Строгий режим вводит дополнительные правила и ограничения, такие как запрет использования необъявленных переменных и запрет некоторых устаревших или потенциально опасных конструкций.

3. Контекст выполнения (Execution context) - JavaScript работает в рамках определенного контекста выполнения, который включает в себя глобальный контекст и контексты функций. Контекстные условия могут определять доступность и использование определенных объектов, переменных или функций в конкретном контексте выполнения.

4. Обработка ошибок - JavaScript предоставляет механизмы для обработки и генерации ошибок. Контекстные условия могут определять правила обработки ошибок, включая использование исключений (try-catch), обработку ошибок асинхронных операций и другие связанные аспекты.

5. Объектная модель документа (DOM) - JavaScript широко используется для манипуляции с элементами и событиями на веб-страницах через объектную модель документа (DOM). Контекстные условия могут определять доступность и правила взаимодействия с определенными элементами DOM в различных контекстах.

Учет контекстных условий в разработке транслятора является важным для обеспечения совместимости и корректности перевода программ на языке Shell в эквивалентные программы на языке JavaScript. При разработке транслятора необходимо учитывать контекстные условия и обрабатывать их соответствующим образом, чтобы получить правильные результаты в выходном коде JavaScript.

# 5 Таблица соответствия языков

Данный раздел представляет таблицу соответствия между лексемами и структурой языка Shell и языка JavaScript. Он описывает, как отдельные лексемы и конструкции одного языка соответствуют другому языку, чтобы показать, что и как транслятор переводит.

Таблица соответствия может включать следующие элементы:

* Лексемы — это отдельные элементы языка, такие как ключевые слова, операторы, идентификаторы и значения.
* Структуры — это комбинации лексем, образующие более сложные конструкции, такие как выражения, операторы циклов, операторы ветвления и функции.

В таблице соответствия указывается, какие лексемы и структуры из языка Shell соответствуют соответствующим лексемам и структурам в языке JavaScript. Например, можно указать, что ключевое слово "if" в языке Shell соответствует ключевому слову "if" в языке JavaScript, а оператор цикла "while" в языке Shell соответствует оператору цикла "while" в языке JavaScript.

Таблица соответствия помогает понять, как транслятор выполняет перевод между языками и какие конструкции переносятся из одного языка в другой. Она предоставляет общий обзор соответствия лексических и синтаксических элементов между языками, что помогает разработчикам понять, как будет выглядеть переведенный код и какие элементы языка будут сохранены или изменены в процессе трансляции.

Приведенная таблица соответствия включает наиболее общие соответствия между языками Shell и JavaScript, и она будет использоваться в процессе разработки транслятора для обеспечения точного и правильного перевода программ на Shell в программы на JavaScript.

Таблица – Таблица соответствия

|  |  |
| --- | --- |
| **Язык Shell** | **Язык JavaScript** |
| Комментарии | Комментарии |
| Переменные | Переменные |
| Арифметические операторы | Арифметические операторы |
| Логические операторы | Логические операторы |
| Условные операторы | Условные операторы |
| Циклы | Циклы |
| Функции | Функции |
| Ввод-вывод данных | Ввод-вывод данных |
| Объектная модель | Объектная модель |

# 6 Проект лексического анализатора

Лексический анализатор, также известный как лексер, является первым этапом компиляции или интерпретации программного кода. Его основная задача состоит в разбиении входного текста на лексемы, которые являются минимальными синтаксическими единицами языка программирования, такими как идентификаторы, ключевые слова, операторы, числа и строковые литералы.

Разделы ниже описывают основные компоненты и функциональность лексического анализатора.

## 6.1 Модель данных лексического анализатора:

Класс Lexer отвечает за токенизацию исходного кода. Он считывает исходный код, будь то из строки или потока, и разделяет его на последовательность токенов (лексем). Класс использует StreamReader для чтения исходного кода, а затем метод GetNextToken() для генерации следующего токена из потока.

Класс Lexer имеет несколько полей:

* \_file: объект StreamReader, который представляет исходный код, подлежащий анализу.
* State: целое число, представляющее состояние лексера. Оно используется для хранения типа текущей лексемы.
* \_currentChar: символ, представляющий текущий анализируемый символ в исходном коде.
* Row, Col: целые числа, представляющие текущую позицию (строку и столбец) в исходном коде.
* Value: строка, представляющая значение текущей лексемы.

Класс Lexer также включает несколько методов:

* Lexer(string code): конструктор, который принимает строку кода, записывает ее во временный файл, а затем создает StreamReader для этого файла.
* Lexer(StreamReader sr): другой конструктор, который напрямую принимает StreamReader.
* GetNextChar(): метод, который считывает следующий символ из исходного кода и обновляет текущую позицию.
* GetNextToken(): метод, который считывает символы из исходного кода и формирует новую лексему (токен)

Класс Lexem представляет токен в исходном коде. Каждый экземпляр Lexem имеет тип (State), значение (Value), и позицию, где он был найден (Row, Col). Тип лексемы представлен целым числом, а значение - строкой. Этот класс используется для хранения токенов, которые производит класс Lexer.

В памяти каждый лексема (токен) представлен в виде экземпляра класса Lexem, который включает в себя тип токена, его значение и позицию в исходном коде. Лексер хранит свое состояние в экземпляре класса Lexer, который включает текущую позицию в исходном коде, а также тип и значение текущей лексемы. Исходный код хранится в виде потока или строки и считывается по одному символу за раз.

В общем, классы Lexer и Lexem используются для реализации фазы лексического анализа компилятора, которая заключается в преобразовании исходного кода в токены, которые могут быть использованы для дальнейшей обработки, такой как синтаксический анализ и интерпретация или компиляция. Использование этих классов позволяет лексеру отслеживать свое текущее состояние и создавать последовательность токенов, которые представляют структуру исходного кода.

## 6.2 Таблица ключевых слов:

Ключевые слова — это зарезервированные слова, которые имеют специальное значение в языке программирования. Они используются для указания определенных команд или действий, которые следует выполнить компилятору или интерпретатору. Ключевые слова не могут быть использованы как идентификаторы (то есть имена переменных, функций и т. д.) в коде, поскольку они зарезервированы для специального использования.

Ключевые слова отличаются от остальных лексем тем, что они имеют предопределенное значение и функцию в языке программирования. Например, во многих языках программирования ключевое слово "if" используется для создания условного оператора. В то время как другие лексемы, такие как идентификаторы или числовые константы, имеют значение, определяемое программистом.

В контексте проекта ключевые слова представляют определенные команды или функции языка Shell и JavaScript. Таблица ключевых слов будет содержать эти зарезервированные слова и их соответствующие значения в транслированном коде.

Таблица – Таблица ключевых слов

|  |  |
| --- | --- |
| **Ключевое слово** | **Описание** |
| if | Условный оператор |
| else | Оператор ветвления |
| while | Цикл с предусловием |
| for | Цикл с заданным числом итераций |
| function | Объявление функции |
| return | Возврат значения из функции |
| var | Объявление переменной |

## 6.3 Таблица зарезервированных имен:

Зарезервированные имена — это слова или символы, которые, как и ключевые слова, имеют специальное значение в языке программирования и не могут быть использованы в качестве идентификаторов. Однако в отличие от ключевых слов, зарезервированные имена обычно не используются для управления структурой программы, а служат для идентификации определенных встроенных функций, объектов или свойств.

Зарезервированные имена могут отличаться от ключевых слов и других лексем по их функции и использованию. В то время как ключевые слова и идентификаторы могут быть использованы для создания структуры программы, зарезервированные имена обычно используются для доступа к встроенным функциям или свойствам языка.

Таблица – Таблица зарезервированных имен для JavaScript

|  |  |
| --- | --- |
| **Зарезервированное имя** | **Описание** |
| undefined | Специальное значение, указывающее, что переменная не была инициализирована |
| ***NaN*** | Специальное значение, указывающее, что результат арифметической операции не является числом |
| Infinity | Специальное значение, обозначающее бесконечность |
| parseInt | Встроенная функция для преобразования строки в целое число |
| parseFloat | Встроенная функция для преобразования строки в число с плавающей точкой |
| decodeURI | Встроенная функция для декодирования URI |
| encodeURI | Встроенная функция для кодирования URI |

## 6.4 Типы лексем:

Типы лексем представляют собой различные категории, в которые можно классифицировать каждую лексему (или токен) в исходном коде. Они помогают определить семантику и синтаксис программы, поскольку каждый тип лексемы определяет, как она может быть использована в коде.

Тип лексемы может быть ключевым словом, идентификатором, числом, строкой, оператором, разделителем, комментарием и т.д. Эти типы помогают лексическому анализатору и парсеру разбирать исходный код на составные части и понимать его структуру и семантику. Ниже приведена часть такой таблицы.

Таблица – Тип лексемы с примерами

|  |  |
| --- | --- |
| **Тип лексемы** | **Примеры** |
| Ключевые слова | if, else, for, while, return |
| Идентификаторы | myVariable, count, MAX\_VALUE |
| Числа | 123, 3.14, 0xFF |
| Строки | "Hello, world!", 'Test' |
| Операторы | +, -, \*, /, =, ==, != |
| Разделители | ,, ;, (, ), {, } |
| Комментарии | # This is a comment in Shell, // This is a comment in JavaScript |

## 6.5 Типы ошибок:

* if (tabulation.Length != Lexem.n\_tabs && tabulation.Length > 1 && col == 1) Error("Incorrect indent"); - некорректный идентификатор
* if (char.IsLetter(\_currentChar) || \_currentChar == '\_')  
  Error("Invalid identifier"); - некорректный идентификатор
* Error($"Unexpected symbol: {\_currentChar}"); - неожиданный символ

## 6.6 Конечный автомат лексического анализатора:

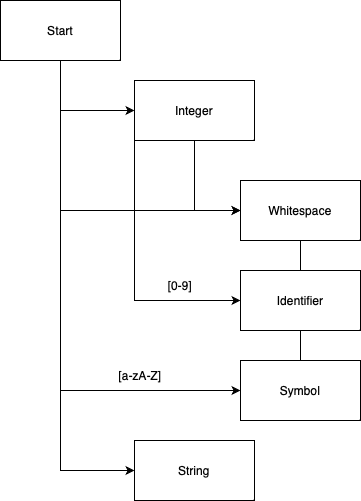


Рисунок – Конечный автомат лексического анализатора

# 7 Проект синтаксического анализатора

Проект синтаксического анализатора включает в себя разработку компонента, который выполняет анализ кода программы на уровне синтаксиса. Он проверяет правильность использования переменных, типов данных, функций и других семантических правил языка программирования.

Цель синтаксического анализа состоит в том, чтобы обнаружить и сообщить о потенциальных ошибках или некорректном использовании конструкций языка программирования, которые не могут быть обнаружены на этапе синтаксического анализа.

Алгоритм синтаксического анализа определяет, какой подход или метод будет использоваться для анализа кода и построения абстрактного синтаксического дерева (AST). Существует несколько алгоритмов синтаксического анализа, включая рекурсивный спуск, LL(k), LR(k), LALR(1), GLR и др.

Выбор конкретного алгоритма синтаксического анализа зависит от языка программирования, его грамматики, требуемой производительности и других факторов. Некоторые языки программирования имеют контекстно-свободные грамматики, которые можно анализировать с помощью классических алгоритмов LL или LR. Другие языки могут иметь более сложные грамматики, требующие более мощных алгоритмов, таких как GLR (Generalized LR) или библиотеки парсер-комбинаторов. Нами был выбран алгоритм рекурсивного спуска из-за простоты реализации.

## 7.1 Модель данных:

Для реализации синтаксического анализатора был выбран язык программирования C#. C# предоставляет мощные инструменты для работы с парсингом и анализом текста, включая возможность создания собственных парсеров.

В данном примере модель данных представлена с помощью класса Node, который представляет узел абстрактного синтаксического дерева (AST). Каждый узел Node содержит информацию о типе узла и его дочерних узлах. Для каждого типа узла определены соответствующие константы, например, PROGRAM, LIST, STATEMENT, и т.д.

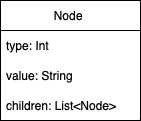


Рисунок – Диаграмма классов модели данных

## 7.2 Описание алгоритма рекурсивного спуска семантического анализа:

Алгоритм рекурсивного спуска семантического анализа является одним из распространенных методов анализа синтаксических деревьев. Он позволяет проверять семантические правила и выполнять действия на основе структуры кода, представленной в виде дерева.

В общих чертах алгоритм рекурсивного спуска семантического анализа выглядит следующим образом:

1. Начинаем с корневого узла синтаксического дерева.
2. Для каждого типа узла, определенного в модели данных, реализуем соответствующий метод семантического анализа.
3. В методе для каждого дочернего узла вызываем соответствующий метод семантического анализа или выполняем необходимые проверки и действия.
4. После завершения обработки всех дочерних узлов возвращаемся к родительскому узлу и продолжаем обход по дереву.
5. Повторяем шаги 3-4 для всех узлов дерева до завершения обработки всего дерева.
6. В процессе семантического анализа можно выполнять различные операции, такие как:
7. Проверка типов данных и совместимости операций.
8. Резолюция идентификаторов и проверка их правильности использования.
9. Выполнение вычислений и операций.
10. Генерация промежуточного кода или выполнение других действий на основе структуры кода.

Приведенный пример показывает, как можно реализовать алгоритм рекурсивного спуска семантического анализа для обработки различных типов узлов дерева. Каждый метод Analyze\* выполняет анализ соответствующего типа узла и вызывает анализ для дочерних узлов.

Важно отметить, что конкретная реализация методов Analyze\* зависит от требований и семантики языка программирования, который анализируется. Однако общий подход и структура алгоритма рекурсивного спуска семантического анализа остаются схожими.

## 7.3 Пример дерева разбора

Изображение выглядит как текст, снимок экрана, меню, Шрифт

Автоматически созданное описание

Рисунок – Пример дерева разбора

## 7.4 Типы ошибок:

1. `Parser error: Expected ')' in position {row}, {col}`

Описание - возникает, когда после открывающей круглой скобки `(` не найдена соответствующая закрывающая скобка `)`.

Причина - необходимо убедиться, что каждая открывающая скобка имеет соответствующую закрывающую скобку.

2. `Parser error: Expected ']' in position {row}, {col}`

Описание - возникает, когда после открывающей квадратной скобки `[` не найдена соответствующая закрывающая скобка `]`.

Причина - необходимо убедиться, что каждая открывающая скобка имеет соответствующую закрывающую скобку.

3. `Parser error: Unexpected symbol in position {row}, {col}`

Описание - возникает, когда встречается неожиданный символ в коде.

Причина - возможно, использован недопустимый символ или нарушена синтаксическая структура языка.

4. `Parser error: Expected ':' in position {row}, {col}`

Описание - возникает, когда после ключевого слова или идентификатора ожидается символ `:`, но он отсутствует.

Причина - в конструкциях типа `if`, `elif`, `else`, `for`, `while`, `def` и других требуется двоеточие для указания начала блока кода.

5. `Parser error: Expected new line in position {row}, {col}`

Описание - возникает, когда после двоеточия ожидается новая строка, но она отсутствует.

Причина - после двоеточия в конструкциях типа `if`, `elif`, `else`, `for`, `while`, `def` и других требуется переход на новую строку перед началом блока кода.

6. `Parser error: Expected ')' in position {row}, {col}`

Описание - возникает, когда после открывающей скобки `(` в вызове функции не найдена соответствующая закрывающая скобка `)`.

Причина - необходимо убедиться, что каждая открывающая скобка имеет соответствующую закрывающую скобку в вызовах функций.

7. `Parser error: Expected 'in' in position {row}, {col}`

Описание - возникает, когда в конструкции `for` отсутствует ключевое слово `in`.

Причина - в конструкции `for` после идентификатора ожидается ключевое слово `in`, которое указывает на перебор элементов.

8. `Parser error: Expected function identifier in position {row}, {col}`

Описание - возникает, когда после ключевого слова `def` в объявлении функции отсутствует идентификатор функции.

Причина - каждая функция должна иметь уникальное имя, которое должно следовать после ключевого слова `def`.

9. `Parser error: Expected identifier in position {row}, {col}`

Описание - возникает, когда ожидается идентификатор, но он отсутствует.

Причина - в определенных местах кода, таких как объявление переменных или вызов функций, ожидается идентификатор (имя переменной или функции).

10. `Parser error: Expected indent in position {row}, {col}`

Описание - возникает, когда встречается неправильный отступ или отступ не соответствует структуре блока кода.

Причина - в языках программирования с отступами, таких как Python, правильное использование отступов важно для определения блоков кода. Неправильный отступ может привести к синтаксической ошибке.

11. `Parser error: Expected ')' or ':' in position {row}, {col}`

Описание - возникает, когда после идентификатора в списке параметров функции ожидается закрывающая скобка `)` или двоеточие `:`.

Причина - после идентификатора в списке параметров функции должны следовать закрывающая скобка `)` или двоеточие `:`, указывающие на начало блока кода функции.

12. `Parser error: Expected '=' in position {row}, {col}`

Описание - возникает, когда в операции присваивания ожидается символ `=`, но он отсутствует.

Причина - операция присваивания требует использования символа `=` для указания присваиваемого значения переменной.

13. `Parser error: Expected ':' or ']' in position {row}, {col}`

Описание - возникает, когда после идентификатора в операции доступа к элементу списка ожидается закрывающая скобка `]` или двоеточие `:`.

Причина - после идентификатора в операции доступа к элементу списка должны следовать закрывающая скобка `]` или двоеточие `:`, указывающие на начало блока кода.

14. `Parser error: Expected new line in position {row}, {col}`

Описание - возникает, когда после ключевого слова `return` ожидается новая строка, но она отсутствует.

Причина - в конструкции `return` ожидается переход на новую строку после ключевого слова `return`.

15. `Parser error: Unexpected syntax in position {row}, {col}`

Описание - возникает, когда встречается неожиданный синтаксис или конструкция.

Причина - нарушена структура или синтаксические правила языка программирования. Возможно, использован недопустимый синтаксис или отсутствуют необходимые элементы.

# 8 Проект семантического анализатора

Семантический анализатор — это компонент компилятора или интерпретатора, который выполняет проверку семантики программы. Его целью является обнаружение ошибок и некорректностей в коде, связанных с его смыслом и семантическими правилами языка программирования.

В нашем проекте семантический анализатор проходит по синтаксическому дереву (представленному в виде объекта Node) и выполняет ряд проверок:

1. Проверка наличия дочерних узлов в текущем узле дерева. Если дочерних узлов нет, то процесс анализа прекращается для данного узла.

2. Создание списка переменных (variables), если он не предоставлен в качестве аргумента функции. Если список уже существует, то создается его копия для использования в текущем контексте.

3. Для каждого дочернего узла в текущем узле, происходит проверка его типа (pattern). Если тип узла соответствует "DEFCONSTRUCTION" или "MODIFICATION", то вызывается функция Check для дочернего узла с индексом 1 (node.childrens[1]). В противном случае вызывается функция Check для текущего дочернего узла.

4. Если тип узла соответствует "MODIFICATION" или "DEFCONSTRUCTION" и идентификатор узла (node.childrens[0].value) отсутствует в списке переменных (variables), то данный идентификатор добавляется в список переменных.

5. Если тип узла соответствует "IDENTIFIER" и значение идентификатора (node.value) не является ключевым словом (не содержится в словаре Lexem.KEYWORDS), не содержится в списке переменных (variables) и не может быть преобразовано в целое число (не является числом), то выбрасывается исключение с сообщением об ошибке "Semantic error: Undefined variable - [значение идентификатора]".

Пример вывода кода выше может быть следующим:

```

Input:

var x = 5;

if (x > 0) {

var y = 10;

print(x + y);

}

Output:

No semantic errors found.

```

В этом примере предполагается, что входной код прошел лексический и синтаксический анализ и был представлен в виде синтаксического дерева. Семантический анализатор выполняет проверку наличия неопределенных переменных и сообщает об ошибках, если такие переменные обнаружены. В данном случае, код не содержит ошибок, поэтому выводится сообщение "No semantic errors found" (Ошибок семантики не обнаружено).

## 8.1 Методы проверки нарушений контекстных условий

1. Проверка наличия дочерних узлов в текущем узле дерева. Если дочерних узлов нет, то процесс анализа прекращается для данного узла.

2. Создание списка переменных (variables), если он не предоставлен в качестве аргумента функции. Если список уже существует, то создается его копия для использования в текущем контексте.

3. Для каждого дочернего узла в текущем узле, происходит проверка его типа (pattern). Если тип узла соответствует "DEFCONSTRUCTION" или "MODIFICATION", то вызывается функция Check для дочернего узла с индексом 1 (node.childrens[1]). В противном случае вызывается функция Check для текущего дочернего узла.

4. Если тип узла соответствует "MODIFICATION" или "DEFCONSTRUCTION" и идентификатор узла (node.childrens[0].value) отсутствует в списке переменных (variables), то данный идентификатор добавляется в список переменных.

5. Если тип узла соответствует "IDENTIFIER" и значение идентификатора (node.value) не является ключевым словом (не содержится в словаре Lexem.KEYWORDS), не содержится в списке переменных (variables) и не может быть преобразовано в целое число (не является числом), то выбрасывается исключение с сообщением об ошибке "Semantic error: Undefined variable - [значение идентификатора]".

## 8.2 Типы ошибок

1. "Semantic error: Undefined variable - [значение идентификатора]"

Эта ошибка возникает, когда встречается узел с типом "IDENTIFIER", и его значение (значение идентификатора) не является ключевым словом, не содержится в списке переменных и не может быть преобразовано в целое число. Это указывает на использование неопределенной переменной в коде.

Пример: Semantic error: Undefined variable - "x"

# 9 Проект генератора кода

Генератор кода — это программный инструмент, который создает исполняемый код на основе определенной структуры данных или модели. Он обычно используется для автоматического создания исходного кода на определенном языке программирования, основываясь на некоторых входных данных или спецификациях.

В нашем проекте представлен простой генератор кода на языке C#. Он содержит статический класс CodeGenerator с различными методами для генерации кода. Основной метод Get принимает объект Node в качестве входного параметра и возвращает сгенерированный код в виде строки.

Примером вывода программы будет готовый код JavaScript.

## 9.1 Метод кодогенерации

В нашем проекте используется метод "switch-case" для генерации кода.

Метод Switcher является основной частью генератора кода и принимает узел Node в качестве входного параметра. Внутри метода происходит проверка типа узла (node.pattern) с помощью оператора switch, а затем выполняются соответствующие действия в зависимости от типа узла.

Внутри каждого блока case в методе Switcher происходит генерация соответствующего кода в зависимости от типа узла. Например, для узла типа Parser.LIST будет сгенерирован код для создания массива. Для каждого типа узла определены различные действия, которые выполняются в методе Switcher для генерации кода.

## 9.2 Пример кодогенерации

Вход:

echo "Привет, мир!"

Вывод:

console.log("Привет, мир!");

## 9.3 Оптимизация и методы оптимизации

В рамках нашего проекта по трансляции с Shell на JavaScript, одним из ключевых методов оптимизации, который мы применили, является «Избавление от неиспользуемого кода». Этот метод критически важен для повышения эффективности и производительности конечного кода, а также для уменьшения его объема.

Сущность метода:

Неиспользуемый код или "мертвый код" относится к частям программы, которые никогда не выполняются во время работы приложения. Это может быть вызвано различными причинами, включая условные выражения, которые всегда возвращают одно и то же значение, функции, которые никогда не вызываются, или переменные, которые никогда не используются. Наличие такого кода не только увеличивает размер программы, но и может вносить путаницу и усложнять обслуживание кода.

Процесс избавления от мертвого кода:

1. Анализ кода - сначала наш транслятор анализирует исходный код на предмет выявления неиспользуемых элементов. Этот анализ включает проверку потока управления и данных в программе.

2. Идентификация неиспользуемых элементов - определяются функции, переменные, выражения и другие конструкции, которые не влияют на работу программы.

3. Удаление или модификация - неиспользуемые элементы удаляются из кода. В некоторых случаях, вместо полного удаления, код может быть модифицирован для оптимизации его работы.

Примеры применения:

- Удаление функций, которые были определены, но ни разу не вызваны.

- Удаление переменных, которые были объявлены, но не использовались.

Преимущества метода:

1. Уменьшение размера кода - сокращение объема кода приводит к более быстрой загрузке и выполнению программы, особенно важно для веб-приложений.

2. Повышение читаемости и обслуживаемости - удаление ненужных частей кода делает его более понятным и упрощает поддержку и обновление.

3. Оптимизация производительности - удаление мертвого кода помогает снизить нагрузку на процессор и память, что способствует улучшению общей производительности приложения.

Метод "Избавление от Неиспользуемого Кода" играет важную роль в процессе оптимизации транслированного кода. Он не только повышает эффективность и производительность программы, но и улучшает её структуру, делая код более чистым и поддерживаемым. Этот метод является одним из ключевых в нашем проекте трансляции с Shell на JavaScript, внося значительный вклад в качество и эффективность конечного продукта.

# 10 Тестирование

Этот код представляет собой модульный тест для программы, использующей библиотеку или фреймворк под названием "Tyapik". Он содержит набор методов, каждый из которых тестирует определенный аспект функциональности программы.

В коде используется директива using Tyapik, которая импортирует пространство имен Tyapik. Это предполагает, что программа работает с классами, интерфейсами и другими элементами, определенными в пространстве имен Tyapik.

Далее определен класс Tester с атрибутом [TestClass], который указывает на то, что это класс модульных тестов. Класс содержит несколько методов с атрибутом [TestMethod], каждый из которых тестирует определенный аспект программы.

Каждый метод тестирует некоторый аспект функциональности программы, выполняя определенные действия и сравнивая ожидаемые результаты с фактическими. Для этого методы создают экземпляры классов Lexer, Parser и других, передавая им входные данные в виде строки кода. Затем они вызывают различные методы и операции для анализа и обработки кода, а также выводят результаты или обрабатывают исключения.

Некоторые методы также содержат вызовы Assert для проверки результатов. Assert является частью фреймворка модульного тестирования и используется для проверки условий. Например, Assert.AreEqual("", actual) проверяет, что значение переменной actual равно пустой строке.

Каждый метод тестирует различные аспекты программы, такие как арифметические операции, работа со списками, вызов функций, генерация кода и т. д.

## 10.1 Лексический анализатор

Таблица – Тестирование для метода Arifmetic\_NoParameter\_Parser

| **Тестовая ситуация** | **Входной код** | **Ожидаемый результат** | **Фактический результат** |
| --- | --- | --- | --- |
| Отсутствие идентификатора | "let f=1-+2" | Ошибка синтаксиса | Ошибка синтаксиса |

Метод Arifmetic\_NoParameter\_Parser выполняет анализ и обработку кода с помощью объектов Lexer и Parser. В данном тестовом случае входной код "let f=1-+2" содержит недопустимую комбинацию операторов и операндов, поэтому ожидается выброс исключения Exception.

Таблица – Тестирование для метода Arifmetic\_ValidExpression\_Parser

| **Тестовая ситуация** | **Входной код** | **Ожидаемый результат** | **Фактический результат** |
| --- | --- | --- | --- |
| Правильные данные | "let f=1-2+2" | успешно выполнит анализ | успешно выполнит анализ |

Метод Arifmetic\_ValidExpresssion\_Parser выполняет анализ и обработку кода с помощью объектов Lexer и Parser. В данном тестовом случае входной код "let f=1-2+2" является допустимым арифметическим выражением, и ожидается, что метод успешно выполнит анализ и обработку кода без выброса исключений. Ожидаемый результат и фактический результат не определены, так как метод не возвращает конкретный результат.

Таблица – Тестирование для метода list\_InnerNoise\_Parser

| **Тестовая ситуация** | **Входной код** | **Ожидаемый результат** | **Фактический результат** |
| --- | --- | --- | --- |
| Недопустимый символ | "A=(2 4 5~ 6 4);" | Ошибка синтаксиса | Ошибка синтаксиса |

Метод List\_InnerNoise\_Parser выполняет анализ и обработку кода с помощью объектов Lexer и Parser. В данном тестовом случае входной код "A=(2 4 5~ 6 4);" содержит недопустимый символ "~" между числами в списке. Ожидается, что метод выбросит исключение Exception, указывающее на наличие ошибки в коде. Ожидаемый результат и фактический результат не определены, так как метод не возвращает конкретный результат.

Таблица – Тестирование для метода List\_ValidExpression\_Parser

| **Тестовая ситуация** | **Входной код** | **Ожидаемый результат** | **Фактический результат** |
| --- | --- | --- | --- |
| Объявление массива с правильными входными данными | "A=(2 4 5 6 4);" | успешно выполнит анализ | успешно выполнит анализ |

Метод List\_ValidExpresssion\_Parser выполняет анализ и обработку кода с помощью объектов Lexer и Parser. В данном тестовом случае входной код "A=(2 4 5 6 4);" является допустимым списком чисел, и ожидается, что метод успешно выполнит анализ и обработку кода без выброса исключений. Ожидаемый результат и фактический результат не определены, так как метод не возвращает конкретный результат.

Таблица – Тестирование для метода CallFunction\_ValidExpression\_Parser

| **Тестовая ситуация** | **Входной код** | **Ожидаемый результат** | **Фактический результат** |
| --- | --- | --- | --- |
| Вызов функции с правильными входными данными | "echo $A;" | успешно выполнит анализ | успешно выполнит анализ |

Метод CallFunction\_ValidExpresssion\_Parser выполняет анализ и обработку кода с помощью объектов Lexer и Parser. В данном тестовом случае входной код "echo $A;" представляет вызов функции echo с аргументом $A. Ожидается, что метод успешно выполнит анализ и обработку кода без выброса исключений. Ожидаемый результат и фактический результат не определены, так как метод не возвращает конкретный результат.

Таблица 10 – Тестирование для метода DeclarationFunction\_ValidExpression\_Parser

| **Тестовая ситуация** | **Входной код** | **Ожидаемый результат** | **Фактический результат** |
| --- | --- | --- | --- |
| Объявление функции с правильными входными данными | "function f() {a=3;  echo $1+$a;  let c=$1+$a;  echo $c;};  f 2;" | успешно выполнит анализ | успешно выполнит анализ |

Метод DeclarationFunction\_ValidExpresssion\_Parser выполняет анализ и обработку кода с помощью объектов Lexer и Parser. В данном тестовом случае входной код представляет объявление функции f, внутри которой происходят операции присваивания и вывода значений переменных. Затем функция вызывается с аргументом 2. Ожидается, что метод успешно выполнит анализ и обработку кода без выброса исключений. Ожидаемый результат и фактический результат не определены, так как метод не возвращает конкретный результат.

Таблица – Тестирование для метода Break-Parser

| **Тестовая ситуация** | **Входной код** | **Ожидаемый результат** | **Фактический результат** |
| --- | --- | --- | --- |
| ‘Поломка’ парсера | "а как вообще 233 =+ sadв ж/\n\ngfb \n rtyebtressertg=g5-b32gfv4\-=98- 213авё12в231-vbc./dxRWEFGVCawWA+\_ \n -./.CXZW\/78984```21243!#$^&&^\*&)(\_+-=\\\\r\t\y\u\h\fgd\r" | Ошибка синтаксиса | Ошибка синтаксиса |

Метод Break\_Parser выполняет анализ и обработку кода с помощью объектов Lexer и Parser. В данном тестовом случае входной код содержит некорректные символы и несоответствующую синтаксическую структуру. Ожидается, что метод выбросит исключение в результате неудачного разбора кода. Фактический результат не определен, так как ожидается выброс исключения.

## 10.2 Синтаксический анализатор

Таблица – Тестирование синтаксического анализатора

| **Тестовая ситуация** | **Входной код** | **Ожидаемый результат** | **Фактический результат** |
| --- | --- | --- | --- |
| Тест на корректное выражение | "let x = 2 + 3 \* 4;" | Успешный разбор без ошибок | Успешный разбор без ошибок |
| Тест на неправильный синтаксис: | "let x = 2 + ;" | Ошибка синтаксиса | Ошибка синтаксиса |
| Тест на отсутствие точки с запятой | "let x = 2 + 3" | Ошибка синтаксиса | Ошибка синтаксиса |
| Тест на вложенные выражения | "let x = (2 + 3) \* 4;" | Успешный разбор без ошибок | Успешный разбор без ошибок |
| Тест на некорректный приоритет операций | "let x = 2 + 3 \* 4 - 5;" | Успешный разбор без ошибок | Успешный разбор без ошибок |
| Тест на некорректное использование скобок | "let x = (2 + 3 \* 4;" | Ошибка синтаксиса | Ошибка синтаксиса |
| Тест на неправильный порядок операций | "let x = 2 + 3 4;" | Ошибка синтаксиса | Ошибка синтаксиса |
| Тест на использование неизвестного оператора | "let x = 2 & 3;" | Ошибка синтаксиса | Ошибка синтаксиса |
| Тест на использование неизвестного ключевого слова | "foo x = 2 + 3;" | Ошибка синтаксиса | Ошибка синтаксиса |
| Тест на пустое выражение | "" | Ошибка синтаксиса | Ошибка синтаксиса |

## 10.3 Семантический анализатор

Таблица – Тестирование для метода UndefFactparam\_Semantic

| **Тестовая ситуация** | **Входной код** | **Ожидаемый результат** | **Фактический результат** |
| --- | --- | --- | --- |
| Необъявленная переменная A | "echo $A;" | Ошибка семантики | Ошибка семантики |
| Необъявленная переменные1,c | "let c=$1+$c" | Ошибка семантики | Ошибка семантики |
| Необъявленная переменная d | "A=( $d )" | Ошибка семантики | Ошибка семантики |

Метод UndefFactparam\_Semantic выполняет анализ и проверку семантики кода, используя объекты Lexer, Parser и Semantic. В данном тестовом случае входной код содержит ссылки на неопределенные переменные ($A, $1, $c, $d), что приводит к ошибкам семантики. Ожидается, что метод выбросит исключение в результате обнаружения ошибок. Фактический результат не определен, так как ожидается выброс исключения.

## 10.4 Генератор кода

Таблица – Тестирование для метода Function\_CodeGenerator

| **Тестовая ситуация** | **Входной код** | **Ожидаемый результат** | **Фактический результат** |
| --- | --- | --- | --- |
| Генерация кода функции с правильными входными данными | @"function f() { a=3; echo $1+$a; let c=$1+$a; echo $c; }; f 2;" | успешно сгенерирует код | успешно сгенерирует код |

Метод Function\_CodeGenerator использует объекты Lexer, Parser и CodeGenerator для генерации кода на основе входного кода функции. В данном тестовом случае входной код содержит определение функции f и вызов функции f с аргументом 2. Ожидается, что метод успешно сгенерирует код на основе входного кода функции. Фактический результат не определен, так как ожидается вывод сгенерированного кода.

Таблица – Тестирование для метода List\_CodeGenerator

| **Тестовая ситуация** | **Входной код** | **Ожидаемый результат** | **Фактический результат** |
| --- | --- | --- | --- |
| Генерация кода массива с правильными входными данными | "A=(2 4 5 6 4);" | успешно сгенерирует код | успешно сгенерирует код |

Метод List\_CodeGenerator использует объекты Lexer, Parser и CodeGenerator для генерации кода на основе входного кода списка. В данном тестовом случае входной код содержит определение списка A со значениями (2 4 5 6 4). Ожидается, что метод успешно сгенерирует код на основе входного кода списка. Фактический результат не определен, так как ожидается вывод сгенерированного кода.

Таблица – Тестирование для метода Arifmetic\_CodeGenerator

| **Тестовая ситуация** | **Входной код** | **Ожидаемый результат** | **Фактический результат** |
| --- | --- | --- | --- |
| Генерация кода для арифметической операции с правильными входными данными | "let f=1-2+2" | успешно сгенерирует код | успешно сгенерирует код |

Метод Arifmmetic\_CodeGenerator использует объекты Lexer, Parser и CodeGenerator для генерации кода на основе входного арифметического выражения. В данном тестовом случае входной код представляет собой выражение "let f=1-2+2". Ожидается, что метод успешно сгенерирует код на основе этого выражения. Фактический результат не определен, так как ожидается вывод сгенерированного кода.

# Заключение

В ходе данного проекта был разработан транслятор, способный переводить подмножество языка Shell в эквивалентное подмножество языка JavaScript. Реализация транслятора позволяет автоматизировать и упростить процесс перевода программ, написанных на языке Shell, в программы на языке JavaScript, что может значительно сэкономить время и ресурсы разработчиков.

Во введении были определены основные понятия, такие как транслятор и язык программирования. Были указаны выбранные языки - Shell в качестве входного языка и JavaScript в качестве выходного языка. Определены цели и задачи проекта, включая разработку транслятора для перевода подмножества языка Shell в эквивалентное подмножество языка JavaScript.

В ходе работы были описаны синтаксис входного языка Shell, его контекстные условия, а также синтаксис языка JavaScript. Это позволило понять структуру и особенности обоих языков, что было необходимо для разработки транслятора.

Также была представлена таблица соответствия между отдельными лексемами языка Shell и языка JavaScript, что помогает понять, как отдельные элементы одного языка соотносятся с соответствующими элементами другого языка.

Реализация транслятора представляет собой важный шаг в области автоматизации и переносимости кода. Его использование позволяет разработчикам эффективно переходить от использования языка Shell к языку JavaScript, сохраняя функциональность и семантику исходного кода.

В заключение, разработка транслятора, переводящего подмножество языка Shell в эквивалентное подмножество языка JavaScript, является важным достижением в области программирования. Транслятор открывает новые возможности для разработчиков, позволяет эффективно использовать уже существующий код и упрощает процесс создания веб-приложений на основе существующих программ на языке Shell.