МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РФ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«ВЯТСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»
ИНСТИТУТ МАТЕМАТИКИ И ИНФОРМАЦИОННЫХ СИСТЕМ
ФАКУЛЬТЕТ АВТОМАТИКИ И ВЫЧИСЛИТЕЛЬНОЙ ТЕХНИКИ
КАФЕДРА ЭЛЕКТРОННЫХ ВЫЧИСЛИТЕЛЬНЫХ МАШИН

Направление 09.03.01 - Информатика и вычислительная техника
(код и наименование направления)
Профиль – Программное и аппаратное обеспечение вычислительной техники

Допускаю к защите
Заведующий кафедрой ЭВМ
_____ / Долженкова М. Л. /
(подпись) *(Ф.И.О.)*

Разработка конструктора Telegram-ботов. Часть 2

Пояснительная записка выпускной квалификационной работы
ТПЖА 09.03.01.331 ПЗ

Разработал: студент гр.ИВТб-4301-04-00 _____ / Крючков И. С. / _____

Руководитель:
к.т.н., доцент, зав. кафедрой ЭВМ _____ / Долженкова М. Л. / _____

Консультант: преподаватель кафедры ЭВМ _____ / Кошкин О. В. / _____

Нормоконтролер: к.т.н., доцент _____ / Скворцов А. А. / _____
(подпись) *(Ф.И.О)* *(дата)*

Киров 2024

Реферат

Крючков И. С. Разработка конструктора Telegram-ботов. Часть 2: ТПЖА 09.03.01.331 ПЗ ВКР / ВятГУ, каф. ЭВМ; рук. Долженкова М. Л.– Киров, 2024. – Гр.ч. 8л. ф.А1; ПЗ 112 с., 53 рис., 12 табл., 11 источников, 4 прил.

КОНСТРУКТОР, ПРЕДМЕТНО-ОРИЕНТИРОВАННЫЙ ЯЗЫК, ГРАММАТИКА, ЛЕКСИЧЕСКИЙ АНАЛИЗ, СИНТАКСИЧЕСКИЙ АНАЛИЗ, СЕМАНТИЧЕСКИЙ АНАЛИЗ, ИНТЕРПРЕТАЦИЯ, TELEGRAM БОТ, GO.

Объект выпускной квалификационной работы – предметно-ориентированный язык.

Цель выпускной квалификационной работы – расширение функциональных возможностей визуального конструктора Telegram ботов за счет создания предметно-ориентированного языка программирования.

Результат работы – разработанный предметно-ориентированный язык и модуль интерпретации для него, расширяющий функциональные возможности визуального конструктора Telegram ботов.

Содержание

Введение

В современном мире стали популярными такие приложения для быстрого и удобного общения как мессенджеры. Таких приложений достаточно много, но большинство пользователей сети интернет все чаще отдают предпочтение мессенджеру Telegram как наиболее удобному, надежному и функциональному. У Telegram имеется функционал чат-ботов. Любой пользователь может создать своего Telegram бота с помощью общедоступного API, который предоставляет методы для управления ботом. Однако, для создания и управления Telegram ботами требуется определенный уровень технических знаний и навыков программирования, что может быть преградой для многих пользователей.

Конструктор Telegram ботов позволяет широкому кругу пользователей создавать программные продукты с помощью визуального редактора. Пользователи могут выстраивать логику работы приложения просто перетаскивая визуальные блоки и соединяя их между собой в логические цепочки. Это значительно упрощает разработку и делает её доступной даже для тех, кто не имеет глубоких знаний в программировании. Однако функциональные возможности такого подхода к построению ботов ограничены набором доступных компонентов, из которых стоится структура бота и их зачастую недостаточно для реализации сложных, специфичных программных продуктов.

Расширение возможностей платформы визуального конструктора возможно за счёт использования предметно-ориентированного языка программирования. С помощью написания инструкций на данном языке, пользователи могут гибко задавать логику работы разрабатываемого бота, тем самым выходить за рамки функционала, предоставляемого стандартными компонентами.

1 Анализ предметной области

В данном разделе проводится анализ предметной области, который позволит обосновать актуальность разработки проекта, приводятся ключевые требования и особенности конструкций, которые должны быть реализованы в предметно-ориентированном языке визуального конструктора Telegram ботов.

1.1 Актуальность разработки

Боты в Telegram являются его популярной особенностью. С их помощью пользователи могут в интуитивно понятной форме выполнять различные действия, не выходя из мессенджера Telegram.

Боты могут обладать различным функционалом и использоваться в различных сферах, например:

- боты для общения с клиентами;
- техническая поддержка;
- продажа товаров и услуг;
- образовательные боты;
- боты для знакомств и общения;
- развлечения;
- утилиты и интрументы.

Боты имеют множетство плюсов как для пользователей, так и для их владельцев, например некоторые из них:

- замена мобильного приложения;
- удобство использования, интерактивное взаимодействие;
- интеграция с другими системами;
- снижение затрат;
- круглосуточный доступ.

С ростом популярности ботов на рынке стали появляться визуальные конструкторы Telegram ботов – решения для разработки и запуска ботов ботов с минимальным написанием программного кода или совсем без него.

Визуальным конструктором называется NoCode/LowCode инструмент, предназначенный для быстрого создания приложений без обязательного знания языков программирования общего назначения. Иными словами, весь процесс разработки – это взаимодействие с визуальными компонентами платформы, с помощью которых выстраивается логика работы приложения. За счет этого конструкторы значительно упрощают и удешевляют разработку и запуск программных продуктов. Ведь не все обладают знаниями и навыками программирования с использованием языков общего назначения, достаточными для создания даже простых программ. Кроме того, при наличии конструктора нет необходимости разрабатывать каждый раз отдельное приложение для выполнения типовых задач, так как конструктор предоставляет необходимый набор инструментов для быстрого создания прототипа [1].

Конструкторы имеют некоторые ограничения, например, при их использовании нельзя выйти за рамки возможностей самого конструктора, а при выходе нового функционала Telegram Bot API, владельцам платформы визуального конструирования ботов потребуется некоторые время на реализацию поддержки новых методов.

Однако использование только визуальных инструментов накладывает некоторые функциональные ограничения. Обычно количество предоставляемых конструктором компонентов невелико и каждый из них способен выполнять только некоторую небольшую функцию, например отправить сообщение. Это значительно ограничивает возможности пользователя в создании уникальных ботов. Чтобы создание Telegram бота было более гибким, в систему можно интегрировать предметно-ориентированный язык программирования, направленный на расширение функциональных возможностей визуального конструктора. В отличие от визуальных блоков – язык позволяет пользователям сервиса гибко описывать логику работы бота.

Предметно-ориентированный язык (domain-specific language, DSL) – это компьютерный язык, специализированный для конкретной предметной области применения [2]. Противоположностью DSL являются языки общего назначения, такие как C++, Python, Go и т.д.

Примеры предметно-ориентированных языков:

– язык запросов SQL – применяется при работе с базами данных;

– shell-скрипты;

– HTML – язык разметки пользовательского веб-интерфейса;

– CSS – каскадные таблицы стилей, описывающие внешний вид веб-страницы.

Предметно-ориентированные языка можно разделить на две группы по способу представления конструкций [3]:

1) текстовые DSL – текстовая форма, по аналогии с языками общего назначения;
2) визуальные DSL – формирование конструкций выполняется в графическом виде.

Визуальные DSL получили большее распространение, поскольку графическое представление информации обладает большей наглядностью.

Также DSL делятся на два типа: внутренние и внешние.

Внутренние языки опираются на язык общего назначения, являются его часть и дополняют его. Синтаксис такого DSL не может нарушать синтаксис базового языка.

Внешние DSL являются самостоятельными языками, имеют свой синтаксис и семантику. Для успешного запуска они имеют свой компилятор или интерпретатор.

DSL разрабатываются с учетом особенностей предметной области, благодаря чему являются менее избыточными по сравнению с языками общего назначения и более понятными для специалистов данной области. Также предметно-ориентированные языки позволяют работать на более высоком уровне абстракции, что увеличивает эффективность решения поставленных задач и снижает необходимость в изучении универсальных языков. DSL языки легче изучать, учитывая их ограниченную область применения.

Помимо приведённых положительных аспектов предметно-ориентированных языков, они имеют некоторые недостатки. DSL по сравнению с языками общего назначения имеют ограниченные возможности, например, малое разнообразие алгоритмов и структур данных. Кроме того, разработка и внедрение предметно-ориентированного языка может привести к значительным

тратам временных и финансовых ресурсов.

1.2 Расширенное техническое задание

В данном разделе представлено техническое задание на разработку предметно-ориентированного языка для конструктора Telegram ботов.

1.2.1 Основание для разработки

Программа разрабатывается на основе учебного плана кафедры «Электронные вычислительные машины» по направлению 09.03.01.

1.2.2 Цель и задача разработки

Целью разработки является создание языка для расширения функциональных возможностей визуального конструктора.

Задача разработки – проектирование и разработки программного продукта по расширению функциональных возможностей визуального конструктора на базе предметно-ориентированного языка.

1.2.3 Краткая характеристика области применения

Программный продукт предоставляет пользователям возможность более гибкого создания приложений в среде визуального построения Telegram ботов.

1.2.4 Назначение разработки

Функциональным назначением является интерпретация кода предметно-ориентированного языка, написанного пользователем визуального конструктора.

Программа-интерпретатор является модулем визуального конструктора и эксплуатируется как его составная часть. Особые требования к конечному пользователю не предъявляются.

1.2.5 Требования к программному продукту

Конструктор ботов делится на клиентскую и серверную части. Серверная часть реализует основной функционал конструктора и логику его работы, а так же предоставляет API интерфейс для клиентской части. Клиентская часть предоставляет собой тонкий клиент в виде пользовательский веб-интерфейса.

Предметно-оринетированный язык должен быть выполнен в виде подключаемого модуля к серверной части платформы. Модуль должен иметь программ-

ный интерфейс для запуска интерпретации кода предметно-оринетированного языка и возврата результата. Помимо передачи кода на интерпретацию необходимо предусмотреть возможность передачи значений внешних переменных, используемых в передаваемом коде.

На клиентской части платформы должен быть реализован визуальный компонент, позволяющий пользователю конструктора писать и редактировать код на предметно-оринтированном языке и указывать внешние переменные, необходимые для успешного запуска кода.

Предметно-оринетированный язык конструктора должен обладать обозначенными ниже характеристиками.

Ключевые конструкции языка: переменные, ветвления, функции.

Поддерживаемые простые типы данных: строки, числа, булевы значения, «Null».

Поддерживаемые составные типы данных: массив, хэш-карта.

Набор возможных операций языка:

– арифметические операции: сложение, вычитание, умножение, деление, операция нахождение остатка от деления;

– логические операции: конъюнкция, дизъюнкция, отрицание;

– операции сравнения: равно, не равно, больше, меньше, больше или равно, меньше или равно;

– управляющие операции;

– вызов функций.

Интерпретатор предметно-ориентированного языка должен поддерживать все вышеперечисленные конструкции.

Необходимо реализовать встроенные функции для работы со строками и массивами. Кроме этого, необходимо разработать специализированные функции, такие как конвертация строки в число и обратно.

Список требуемых встроенных функций:

– определение длины строки или массива;

– добавление элемента в конец массива;

– получение первого элемента массива;

– получение последнего элемента массива;

– конвертация целого числа в строку и обратно.

В задачи интерпретатора входит выполнение следующих этапов:

– лексический анализ;
– синтаксический анализ;
– семантический анализ;
– исполнение операций;

Исходными данными является код программы на предметно-ориентированном языке.

Выходными данными является значение, полученное в результате успешного исполнения кода.

В случае возникновения ошибки на каком-либо из этапов интерпретации программа должна возвращать человекочитаемую ошибку.

1.2.6 Требования к надежности

Программа должна функционировать в соответствии с заданными требованиями при отсутствии сбоев технических средств.

Вывод

В данном разделе был проведен анализ предметной области, в результате чего было определено, что визуальные конструкторы значительно упрощают процесс создания и запуска Telegram ботов, однако, они имеет функциональные ограничения, которые препятствуют разработке продукта со сложной логикой работы. Предметно-ориентированный язык позволяет расширить возможности визуального конструктора и создавать уникальных ботов, в соответствии с пользовательскими потребностями.

Также было рассмотрено расширенное техническое задание, в котором были определены цели и задачи разработки и основные требования к разрабатываемому программному продукту.

Основываясь на этом можно приступить к разработке предметно-ориентированного языка для визуального конструктора Telegram ботов, так как рассмотренная проблема является актуальной.

2 Разработка архитектурно-структурных решений

Выполнение кода предметно-ориентированного языка осуществляется за счет интерпретатора, который должен быть спроектирован в виде модуля, подключаемого к серверной части конструктора Telegram ботов.

Обобщенная модульная структура серверной части конструктора представлена на рисунке 1.

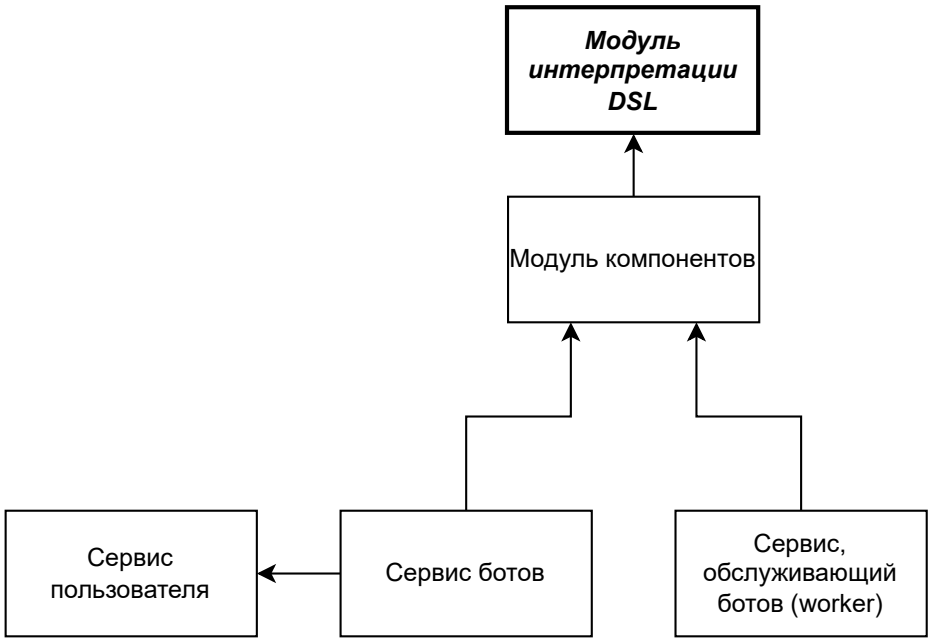

Рисунок 1 – Модульная структура серверной части конструктора

Сервис ботов отвечает за управление состоянием бота и редактирование его компонентной структуры. Сервис ботов зависит от сервиса пользователей, который предоставляет первому методы для авторизации пользователя. Также сервис ботов зависит от модуля компонентов, который описывает структуры компонентов и реализует их логику выполнения.

Обслуживающий сервис отвечает за логику работы бота. Он выполняет обработку запроса к боту от пользователя Telegram. В соответствии с разработанной компонентной структурой бота, обслуживающий сервис вызывает методы модуля компонентов для запуска их логики выполнения.

Одним из компонентов является компонент выполнения кода на предметно-

ориентированном языке. Отсюда, модуль компонентов зависит от модуля интерпретации предметно-ориентированного языка. При запуске компонента выполнения DSL кода, выполняется вызов функции выполнения кода из модуля интерпретации.

В данном проекте для возможности интеграции с серверной частью конструктора Telegram ботов необходимо реализовать программный интерфейс, который бы предоставлял возможность передачи кода и значений внешних переменных, интерпретатору предметно-ориентированного языка.

Работа интерпретатора состоит из последовательного выполнения следующих этапов:

1) лексический анализ;
2) синтаксический анализ;
3) семантический анализ;
4) исполнение команд.

Обобщенная структура интерпретатора представлена на рисунке 2.

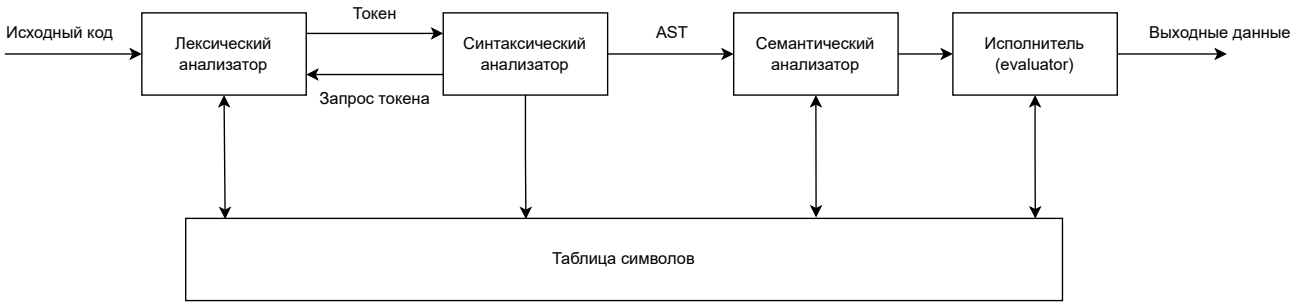

Рисунок 2 – Обобщенная структура интерпретатора

Для решения поставленной задачи, в первую очередь, необходимо в соответствии с указанными требованиями разработать грамматику предметно-ориентированного языка и спроектировать обобщенную структуру программы интерпретатора.

2.1 Разработка грамматики языка

Описание языка программирования основывается на теории формальных языков. В данном разделе проводится разработка формальной грамматики языка и её описание.

2.1.1 Способы задания языков

Для задания языка можно воспользоваться следующими методами:

1) перечислить все цепочки языка;
2) указать способ порождения цепочек;
3) определить метод распознавания допустимых цепочек.

Перечисление всех цепочек языка возможно в исключительных случаях, например, когда для управления некоторой системой достаточно двух-трех команд.

Механизм порождения цепочек предполагает использование формальной порождающей грамматики.

Формальная порождающая грамматика – это математическая система, описывающая правила построения цепочек некоторого (формального) языка [4].

Распознавание допустимых цепочек осуществляется с помощью некоторого логического устройства – распознавателя. На вход распознавателя подается цепочка, а на выходе образуется логическое значение «истина» в случае принадлежности цепочки языку и «ложь», если цепочка языку не принадлежит. Распознаватели строятся на основе теорий конечных автоматов и автоматов с магазинной памятью.

Методы порождения и распознавания тесно связаны. Механизм порождения обычно используется при описании языка, а распознаватель при его реализации, т.е. в трансляторе.

Описать синтаксис языков программирования можно несколькими способами, например, такими как формы Бэкуса-Наура, диаграммы Вирта и другими. Это методы задают правила вывода, определяющие возможные конструкции цепочек языка. В данном проекте для описания грамматики языка используется расширенная форма Бэкуса-Наура (РБНФ).

2.1.2 Применение расширенной формы Бэкуса-Наура для описания формальной грамматики языка

Расширенная форма Бэкуса-Наура – формальная система определения синтаксиса, в которой одни синтаксические категории последовательно определяются через другие. Используется для описания контекстно-свободных грамматик [5].

Формальная грамматика задаётся четвёркой вида:

$G = (V_T, V_N, P, S)$,

где V_T – множество терминальных символов грамматики – конечные элементы языка, не разбирающиеся на более мелкие составляющие в рамках синтаксического анализа, например ключевые слова, цифры, буквы латинского алфавита.

V_N – конечное множество нетерминальных символов – элементов грамматики, имеющих собственные имена и структуру. Каждый нетерминальный символ состоит из одного или более терминальных и/или нетерминальных символов.

P – множество правил вывода грамматики.

S – начальный символ грамматики, $S \in V_N$ [4].

РБНФ является одним из способов представления формальных грамматик. РБНФ состоит из множества правил вывода, каждое из которых определяет синтаксис некоторой конструкции языка.

Некоторые основные конструкции РБНФ:

– A, B – конкатенация элементов;
– A | B – выбор (A или B);
– [A] – элемент в квадратных скобках может отсутствовать (аналог - «?»);
– {A} – повторение элемента 0 или более раз (аналог - «*»);
– (A B) – группировка элементов;
– (* … *) – комментарий;
– «;» – отмечает окончание правила (аналог - «.»).

Кроме того, в качестве синтаксического сахара могут использоваться следующие символы:

– «*» - предыдущий элемент может встречаться 0 или более раз;
– «?» - предыдущий элемент является необязательным (присутствует 0 или 1 раз);

– «+» - предыдущий элемент встречается 1 или более раз.

В соответствии с данными правилами описание синтаксиса предметно-ориентированного языка будет выглядеть следующим образом:

(* Точка входа *)

Program = Statement+

(* Основные конструкции *)

Statement = AssignStmt | FunctionDecl | ExpressionStmt | ReturnStmt | BlockStmt | IfStmt .

ExpressionStmt = Expression .

AssignStmt = Identifier assign_op Expression .

ReturnStmt = ”return” [Expression] .

BlockStmt = ”{” StatementList ”}”.

StatementList = { Statement ”;” } .

IfStmt = ”if(” [Expression] ”)” BlockStmt [”else” BlockStmt] .

(* Выражения *)

Expression = UnaryExpr | Expression binary_op Expression .

UnaryExpr = PrimaryExpr | unary_op UnaryExpr .

PrimaryExpr = Operand | PrimaryExpr Index | CallExpr .

Index = ”[” Expression ”]” .

ExpressionList = Expression { ”,” Expression } .

(* Определение структуры функции *)

FunctionDecl = ”fn(” [ParameterList] ”)” BlockStmt .

ParameterList = Identifier { ”,” Identifier } .

Arguments = ”(” [ExpressionList] ”)” .

CallExpr = Identifier Arguments .

(* Массив *)

Array = ”[” [ExpressionList] ”]” .

(* Хэш-карта *)
Key = stringLiteral | intLiteral | Identifier | Expression .
KeyedElement = [Key ”:” Expression] .
Map = ”{” KeyedElement { ”,” KeyedElement } ”}” .

Identifier = (letter | ”_”) letter | ”_” | digit .
Operand = Literal | ”(” Expression ”)” .
Literal = intLiteral | stringLiteral | Array | Map .
intLiteral = digit { digit } .
stringLiteral = « ” » { ascii_char } « ” » .

(* Операторы *)
binary_op = ”||” | ”&&” | rel_op | add_op | mul_op .
rel_op = ”==” | ”!=” | ¨<¨| ”<=” | ”>” | ”>=” .
add_op = ”+” | ”-” .
mul_op = ”*” | ”/” | ”%” .
assign_op = ”=” .
unary_op = ”-” | ”!” .

(* Примитивы (цифры, буквы) *)
digit = ”0” ... ”9” .
letter = ”A” ... ”Z” | ”a” ... ”z” .
ascii_char = ascii character .

Начальное состояние, с которого начинается разбор – Program.

2.2 Разработка лексического анализатора

Лексический анализ – процесс разбора входной последовательности символов на распознанные группы – лексемы.

Лексемой является структурная (минимальная значимая) единица языка, состоящая из элементарных символов языка и не содержащая в своём составе других структурных единиц языка [6].

В ходе выполнения лексического анализатора каждая лексема идентифицируется и преобразуется в токен.

Токен – экземпляр лексемы, представляющий собой пару «тип лексемы» и «значение». «Тип» указывает на принадлежность лексемы к определенной категории, например, идентификатор, число и т.д., а «значение» содержит конкретные данные, соответствующие этой лексеме, оно понадобится на дальнейших этапах.

Категории токенов, которые используются в разрабатываемом предметно-ориентированном языке:

- идентификаторы;
- числа;
- строки;
- разделители;
- операторы (арифметические, сравнения и т.д);
- скобки;
- специальные (конец входной последовательности и т.п)
- ключевые слова.

Полный список токенов с указанием категории и примерами лексем приведен в таблице 1.

Процесс лексического анализа является первым шагов в трансляции исходного кода программы и формирует основу для следующих этапов, таких как синтаксический анализ и построение абстрактного синтаксического дерева.

Таблица 1 – Токены с примерами

Токен	Категория	Пример лексемы
IDENT	Идентификатор	qwe
INT	Число	123
STRING	Строка	«привет, hello»
ASSIGN	Оператор	=
PLUS	Оператор	+
MINUS	Оператор	-
STAR	Оператор	*
SLASH	Оператор	/
EXCLAMINATION	Оператор	!
PERCENT	Оператор	%
EQ	Оператор	==
NEQ	Оператор	!=
LEQ	Оператор	<=
GEQ	Оператор	=>
LT	Оператор	<
GT	Оператор	>
LAND	Оператор	&&
LOR	Оператор	||
COMMA	Разделитель	,
SEMICOLON	Разделитель	;
LPAR	Скобка	(
RPAR	Скобка	)
LBRACE	Скобка	{
RBRACE	Скобка	}
LBRACKET	Скобка	[
RBRACKET	Скобка	]
IF	Ключевое слово	if
ELSE	Ключевое слово	else
TRUE	Ключевое слово	true
FALSE	Ключевое слово	false
FUNC	Ключевое слово	fn
RETURN	Ключевое слово	return
ILLEGAL	Специальный	@
EOF	Специальный	Конец файла

Схема взаимодействия лексического и синтаксического анализаторов показана на рисунке 3.

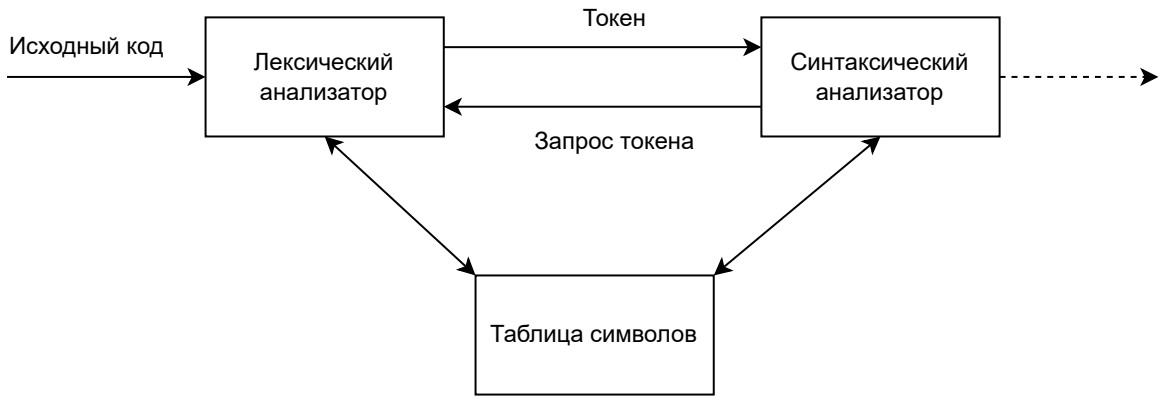

Рисунок 3 – Схема взаимодействия лексического и синтаксического анализаторов

При запросе нового токена лексический анализатор считывает входной поток символов до точной идентификации следующего токена.

Процесс распознавания токенов из входного потока символов языка можно показать с помощью диаграмм переходов состояний.

На рисунке 4 показана диаграмма для определения токенов «=» и «==».

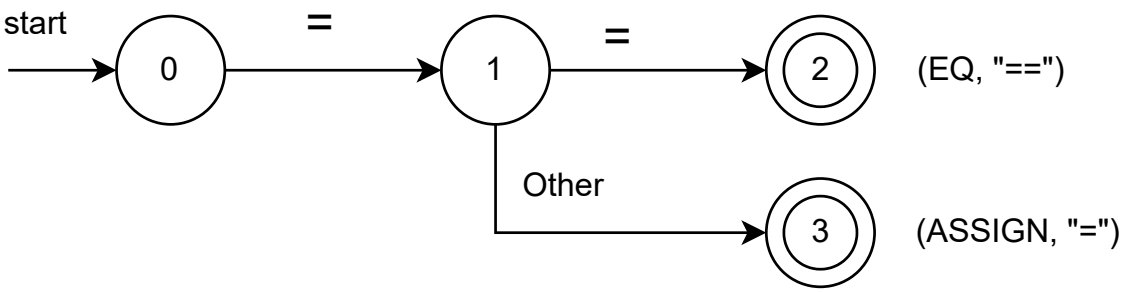

Рисунок 4 – Диаграмма переходов для определения «=» и «==»

Работа начинается с состояния 0, в котором считывается следующий символ из входного потока. Если полученный символ «=», то по дуге, помеченной «=» выполняется переход в состояние 1. В состоянии 1 выполняется считывание

следующего символа. Если этот символ «=», выполняется переход в состояние 2 – заключительное состояние, в котором найден токен «EQ», в том случае, если был получен символ отличный от «=», происходит переход по дуге «other» в состояние 3 с токеном «ASSIGN».

Диаграмма для распознавания целого числа представлена на рисунке 5.

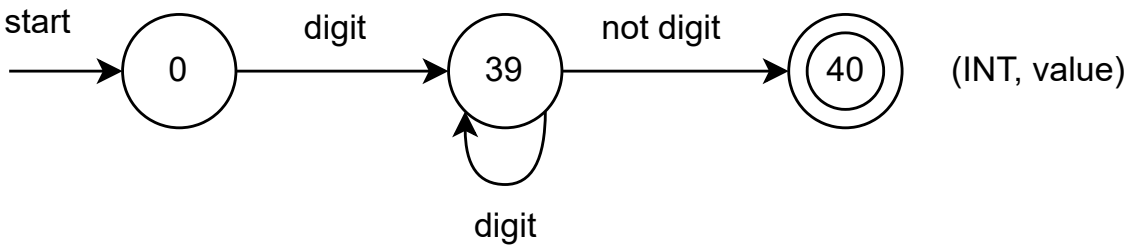

Рисунок 5 – Диаграмма переходов для определения целого числа

При получении в начальном состоянии цифры, выполняется переход в состояние 39, в котором автомат находится до тех пор, пока не получит на вход символ, отличный от цифры, при получении такого символа выполняется переход в конечное состояние 40. По мере определения очередной цифры, она заносится в буфер. В состоянии 40 возвращается токен INT и значение числа из буфера.

Считывание ключевых слов и идентификаторов показано с помощью диаграммы передов на рисунке 6.

Из начального состояния происходит переход в состояние 35, если была получена буква. По аналогии с состоянием 39 выполняется циклическое считывание букв с занесением в буфер. Если была получена не буква, выполняется переход в состояние 36, в котором проверяется принадлежность считанной строки к списку ключевых слов. В случае, если считанная строка является ключевым словом, автомат переходит в завершающее состояние 37 в котором указывается тип токена для полученного ключевого слова и значение. Если в состоянии 36 проверка показала, что строка на является ключевым словом, то выполняется переход в состояние 38 – определен идентификатор.

Полная диаграмма переходов состояний представлена на рисунке 7

Процесс распознавания токена начинается с начального состояния 0. В за-

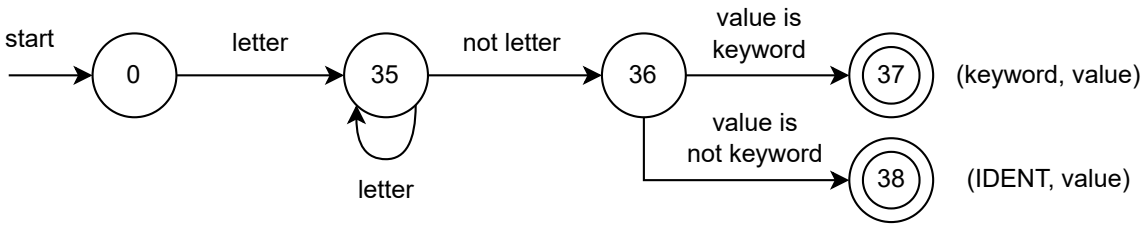

Рисунок 6 – Диаграмма переходов для определения идентификаторов и ключевых слов

висимости от полученного символа выполняется переход в конкретное состояние. Однако, если в начальном состоянии был получен символ, для которого нет дуги, по которой он бы мог перейти в определенное для него состояние, выполняется переход по дуге «other» в состояние 42 с определением токена ILLEGAL. После определения очередного токена в конечном состоянии, автомат начинает работу заново с начального состояния. Считывание входного потока символов прекращается при поступлении нулевого символа (null character) с определением токена EOF. Вместе с токеном, лексический анализатор возвращает его позицию в входном коде.

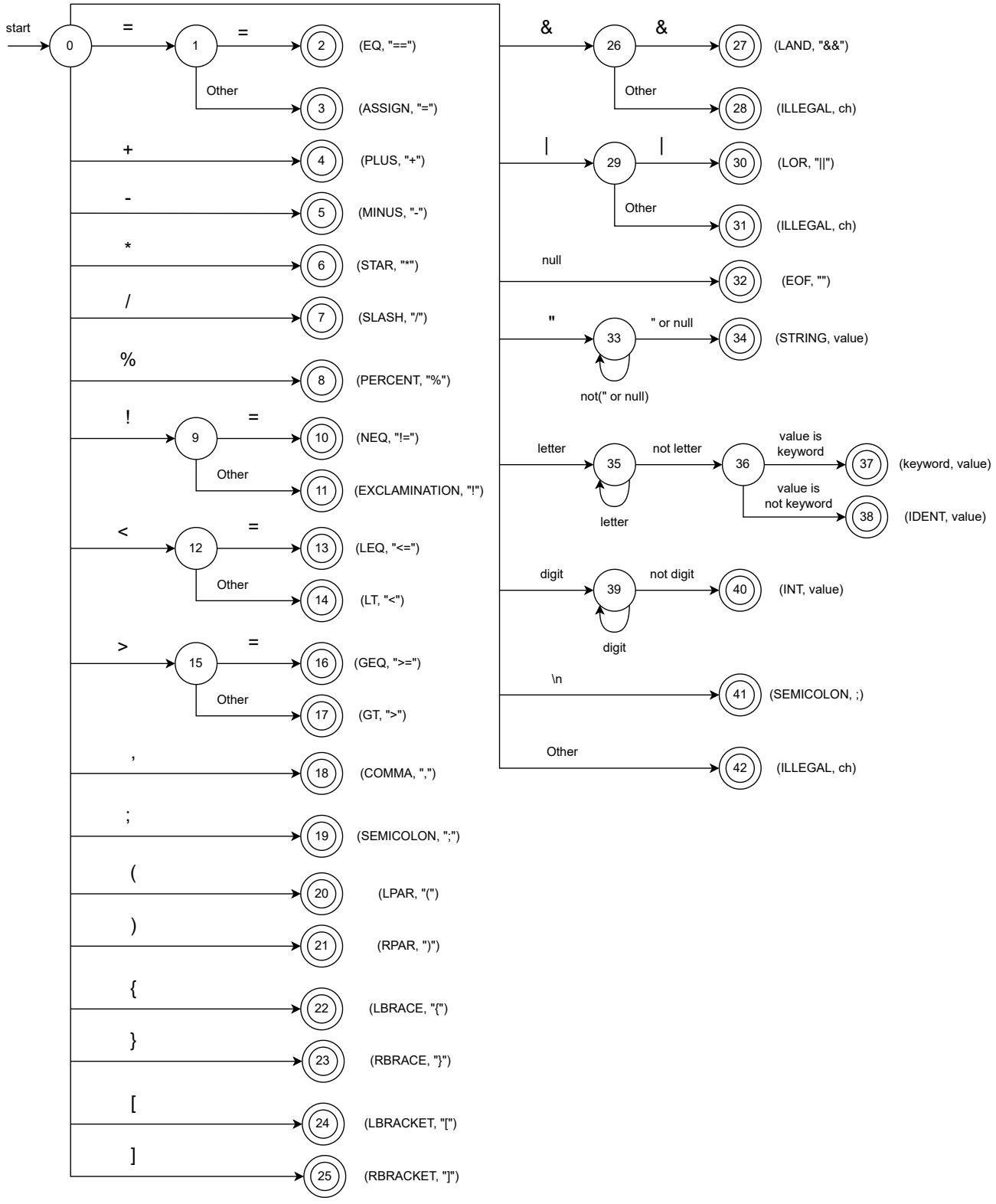

Рисунок 7 – Полная диаграмма переходов состояний

2.3 Разработка синтаксического анализатора

Синтаксический анализ – процесс сопоставления последовательности токенов с формальной грамматикой языка. Результатом работы синтаксического анализатора является абстрактное синтаксическое дерево (AST), которое отражает синтаксическую структуру входной последовательности и содержит всю необходимую информацию для дальнейших этапов работы транслятора [7].

В задачу синтаксического анализа входит поиск и выделение основных синтаксических конструкций текста входной программы, установление типа и проверка правильности каждой синтаксической конструкции, а так же представление их в виде AST [8].

Существует два основных метода синтаксического анализа:

– нисходящий;
– восходящий.

В данном проекте реализован нисходящий анализатор, работающий по методу рекурсивного спуска, известный как парсер Пратта, который впервые описал Вон Пратт в статье «Нисходящий парсер с операторным предшествованием».

Этот метод основан на идее приоритета операторов и обработке различных уровней приоритета в выражениях. В парсере Пратта каждый оператор имеет свой уровень приоритета. Операторы с более высоким приоритетом связываются с операндами сильнее, чем операторы с более низким приоритетом. Значения приоритетов для каждого оператора разрабатываемого предметно-ориентированного языка показаны в таблице 2.

Таблица 2 – Приоритеты операторов

Приоритет	Операторы
0	Минимальный приоритет
1	=
2	||
3	&&
4	== !=
5	< > <= >=
6	+ -
7	* / %
8	-x or !x
9	(
10	[

Для примера работы приоритета операторов рассмотрим пример построения AST для арифметического выражения: $3 + 1 * 4 * 6 + 8$. Диаграмма, показывающая рекурсивные вызовы для формирования выражений к приведенному примеру изображена на рисунке 8.

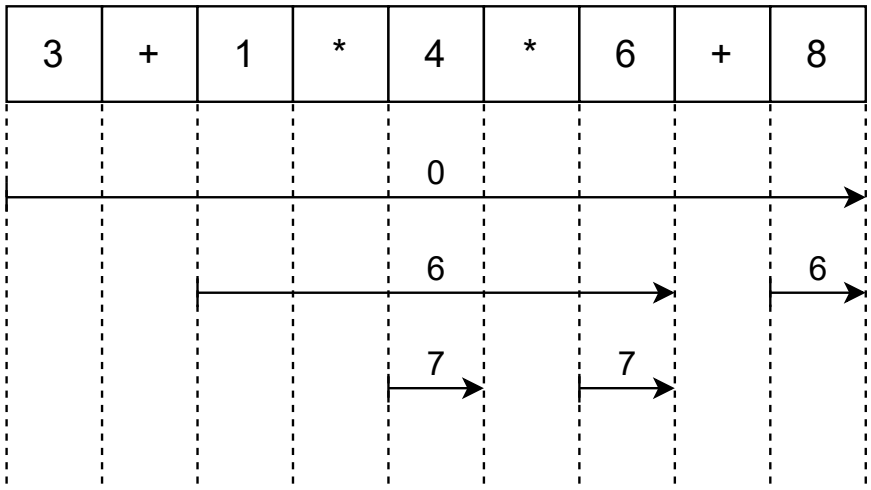

Рисунок 8 – Диаграмма рекурсивных вызовов

Над стрелками обозначены приоритеты операторов, к которым относится эта стрелка. В самом начале распознавания выражения значение приоритета рав-

няется 0. По мере обнаружения оператора, приоритет которого выше текущего алгоритм переходит на следующий уровень рекурсии. По диаграмме видно, что справа от первого оператора «+» длинная стрелка с приоритетом 6, группирует члены умножения, так как операция умножения имеет больший приоритет, чем сложение. Эта стрелка заканчивается перед последним «+», так как приоритет оператора, относящегося к этой стрелке не ниже приоритета последнего «+». Другими словами, экземпляр выражения с более низким приоритетом ожидает результата формирования выражения с более высоким приоритетом.

Абстрактное синтаксическое дерево для данного примера приведено на рисунке 9.

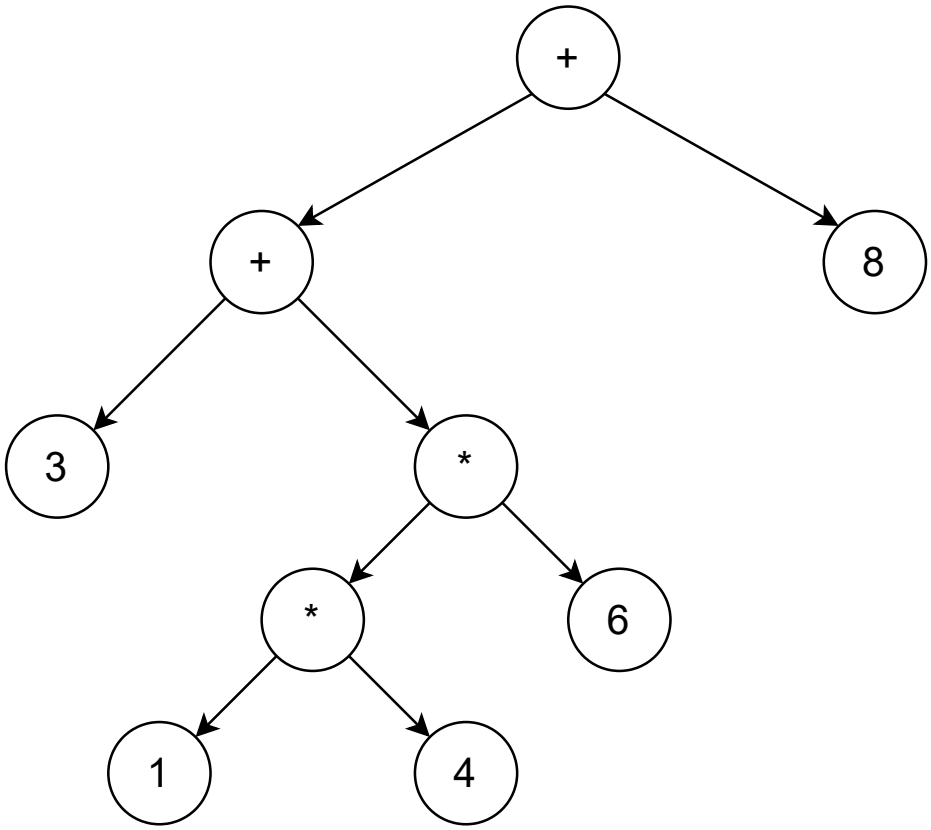

Рисунок 9 – Абстрактное синтаксическое дерево

Некоторые схемы алгоритма построения абстрактного синтаксического дерева представлены на рисунках 10 - 12.

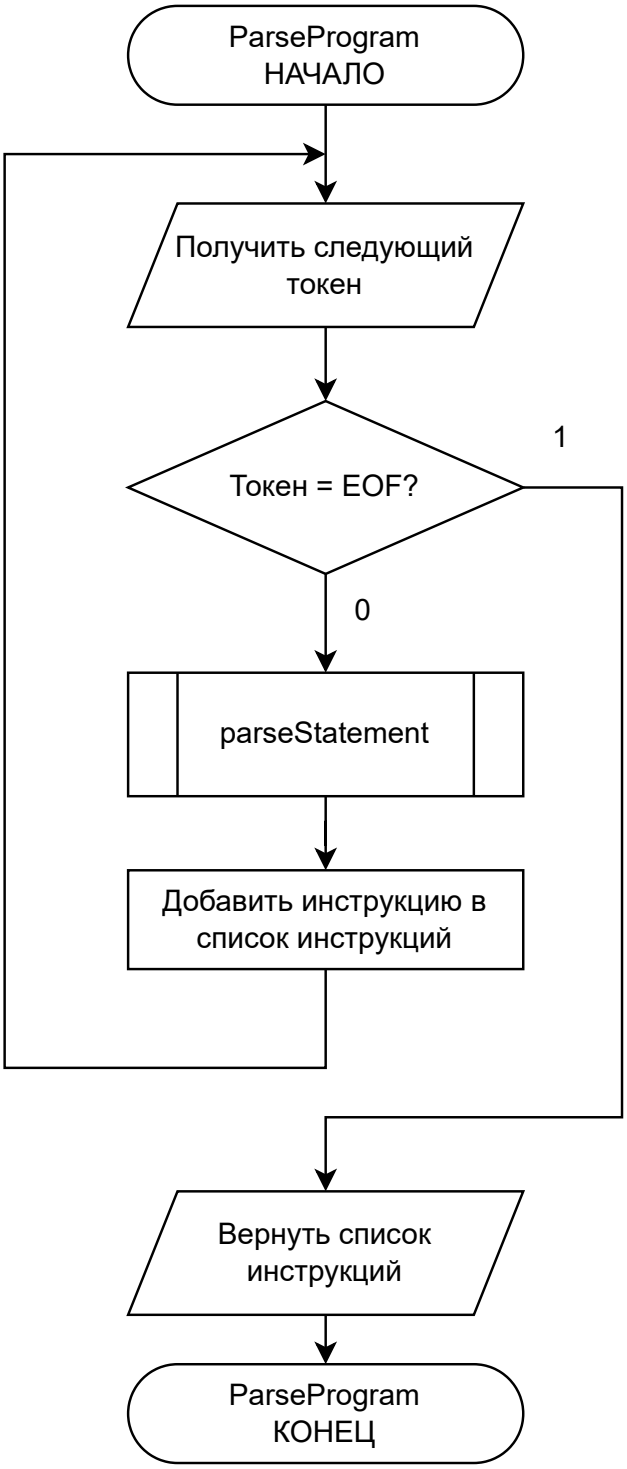

Рисунок 10 – Схема алгоритма «ParseProgram»

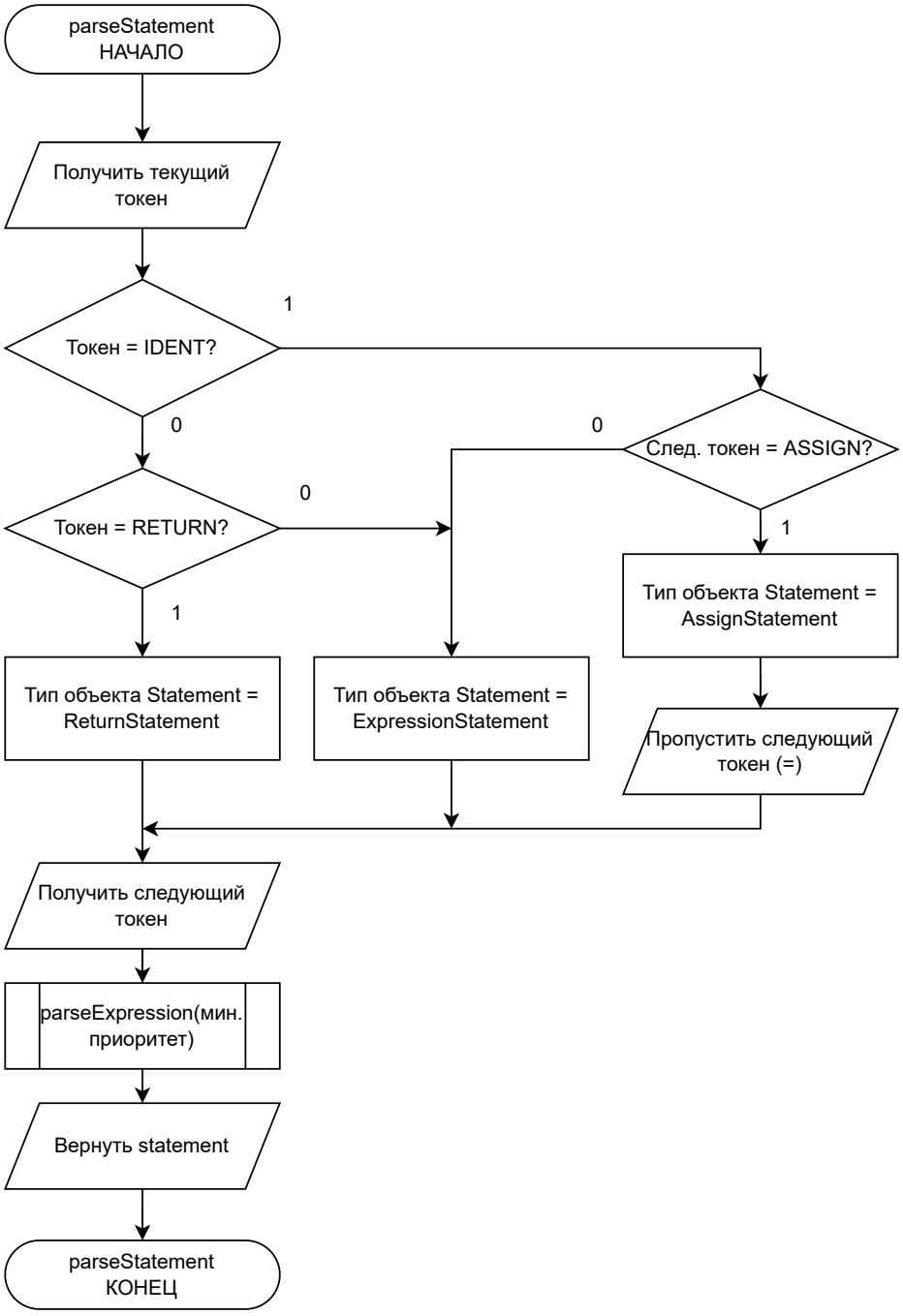

Рисунок 11 – Схема алгоритма «parseStatement»

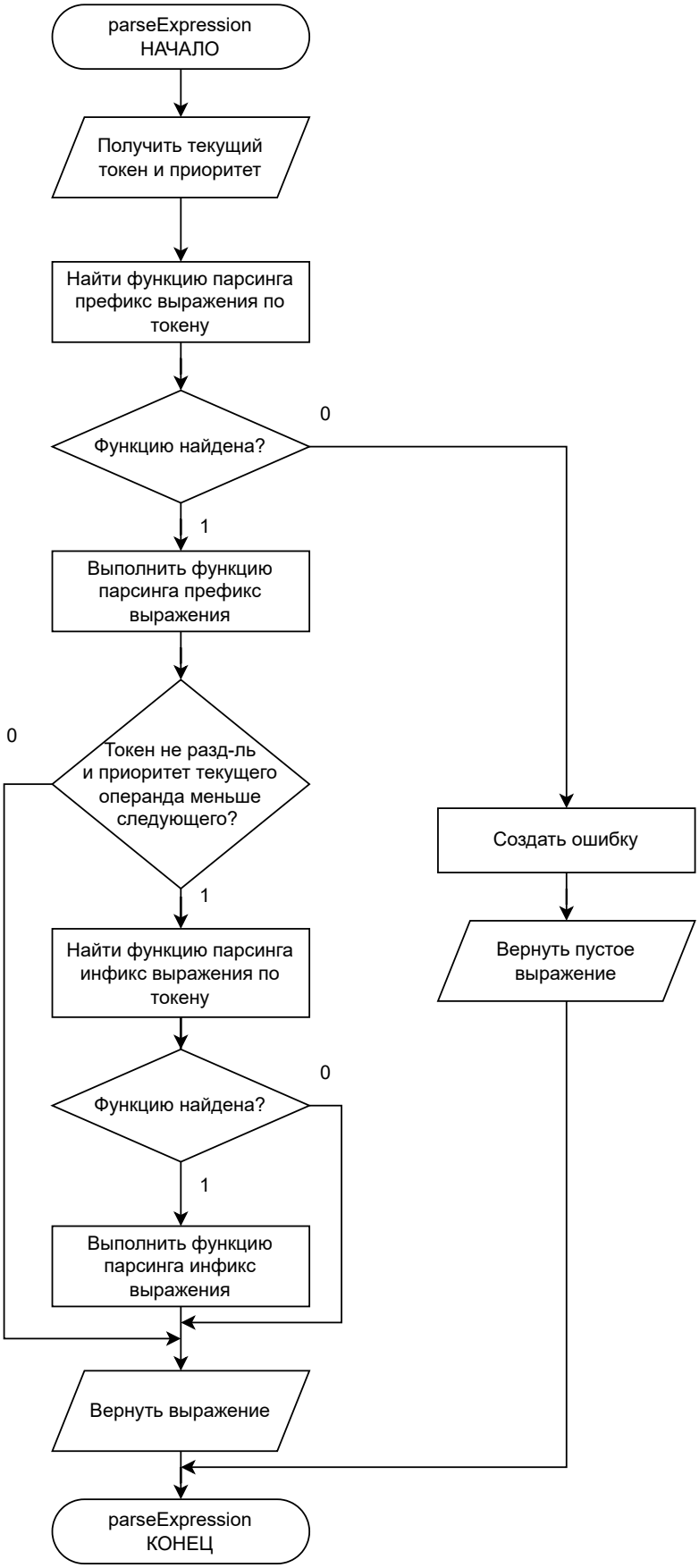

Рисунок 12 – Схема алгоритма «parseExpression»

2.4 Разработка семантического анализатора

Семантический анализ – это этап работы транслятора, во время которого выполняется проверка текста исходной программы с точки зрения семантики языка. В ходе семантического анализа выполняется такие операции, как проверка типов данных, правильность использования переменных, функций, выражений, а также обнаружение смысловых ошибок в коде [7].

Обобщенная модульная структура серверной части конструктора рассмотрена ранее, см. рисунок 1.

Процесс семантического анализа выполняется после построения абстрактного синтаксического дерева синтаксическим анализатором. Семантический анализ сгруппирован с этапом исполнения выражений. Таким образом при одном проходе AST алгоритм выполняет проверку узла дерева на семантическую корректность и переход к его исполнению, если не было обнаружено ошибок при семантическим анализе. В случае обнаружения семантической ошибки возвращается информация о ней, а процесс интерпретации переход к следующему выражению.

На этапе проектирования семантического анализатора закладываются решения, которые лягут в основу принципов написания кода на разрабатываемом языке. Например, на этапе построения семантического анализатора, нужно решить, как будет обрабатываться значение в условном выражении, не имеющее тип boolean. Есть несколько вариантов обработки таких значений. Один из них - всегда принимать данное значение за «false» и идти в соответствующую ветвь. В ином случае, при обнаружении значения с типом, отличным от boolean необходимо вернуть ошибку и прекратить выполнение данного выражения. В этом проекте будет использован второй вариант с возвратом ошибки.

Семантический анализатор принимает на вход элементы абстрактного синтаксического дерева. После проверки семантической корректности выражения AST, следует его вычисление. Результаты вычислений выражений, как промежуточные, так и окончательные необходимо каким-то образом представить в памяти. Это необходимо в первую очередь для получения ранее вычисленных выражений

и работы с ними. В качестве примера можно рассмотреть код, представленный на рисунке 13.

```
X = 5;
X + 3
```

Рисунок 13 – Пример кода использования объявленной переменной

В первой строке присваивается значение 5 переменной «X». Затем выполняется выражение «X + 3». Чтобы получить значение данного выражения нужно получить ранее вычисленное значение 5. Для этого необходимо как-то сохранить его в памяти.

Решение данной задачи состоит в введении внутреннего представления вычисленных значений на время семантического анализа и этапа исполнения. Примем некоторую объектную систему, состоящую из набора объектов, каждый из которых будет содержать информацию о представляемом им типе данных в предметно-ориентированном языке.

Каждый объект должен содержать информацию о значении представляемого им типа данных предметно-ориентированного языка. Кроме этого, необходимо реализовать возможность определения того, какой тип данных предметно-ориентированного языка представляет объект, а также функционал получения строкового значения объекта. Кроме этого, типы данных: целое число, строка и булево значение могут использоваться в виде ключей в хэш-карте. Для этого необходимо специально для объектов, представляющих эти типы предусмотреть функцию вычисления хэш строки от их значения.

Объектная система должна представлять все типы данных предметно-ориентированного языка, а именно: целые числа, строки, булевы значения, массивы, хэш-карты. Также необходимо ввести дополнительные объекты, представляющие семантические ошибки, значение «null», функции, операцию возврата значения из функции.

Объектная система может быть представлена в виде диаграммы классов. Диаграмма классов представлена на рисунке 14.

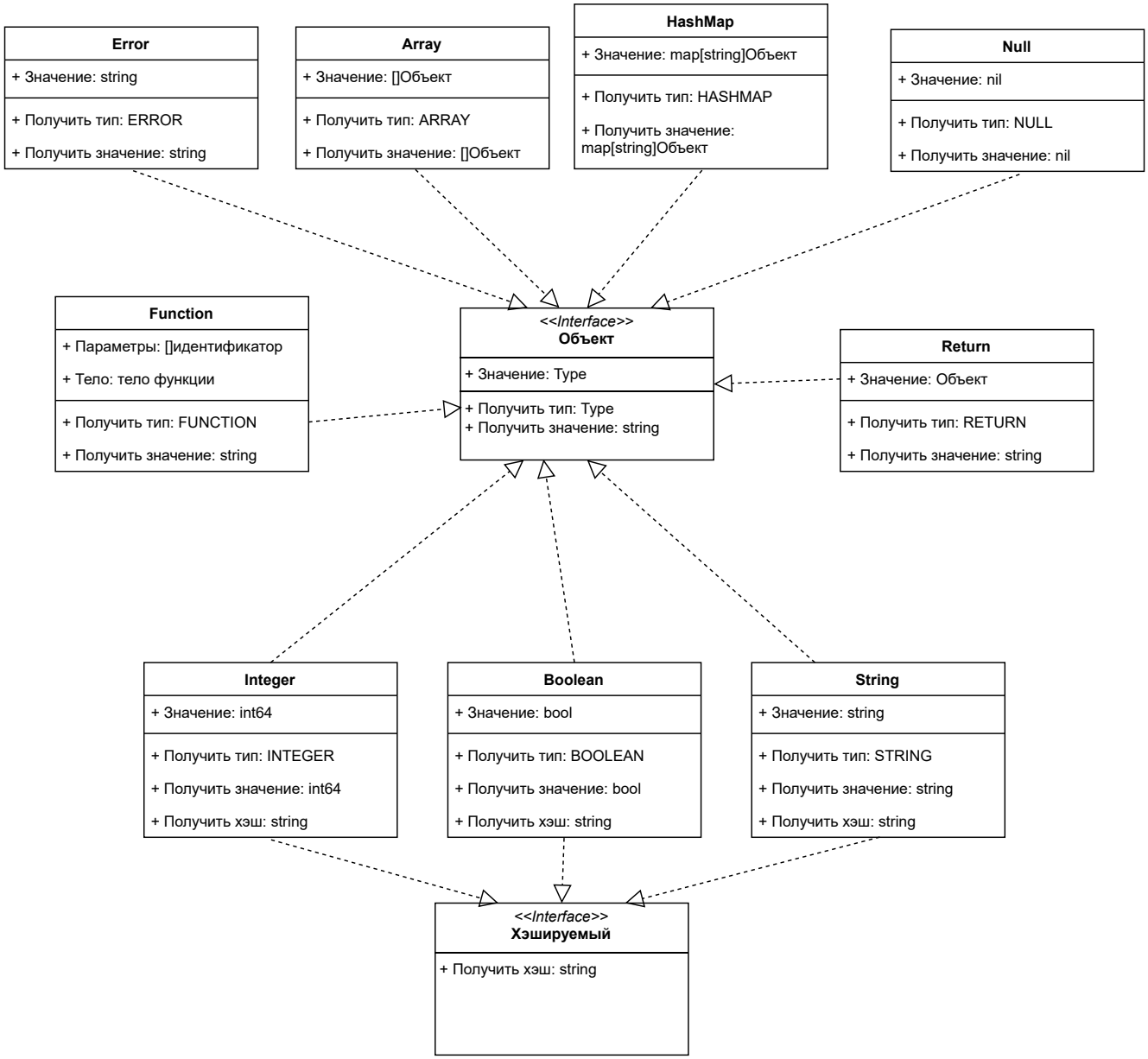

Рисунок 14 – Диаграмма классов

В качестве примера работы алгоритма на рисунках 15 - 19 приведены схемы алгоритмов семантического анализа некоторых выражений.

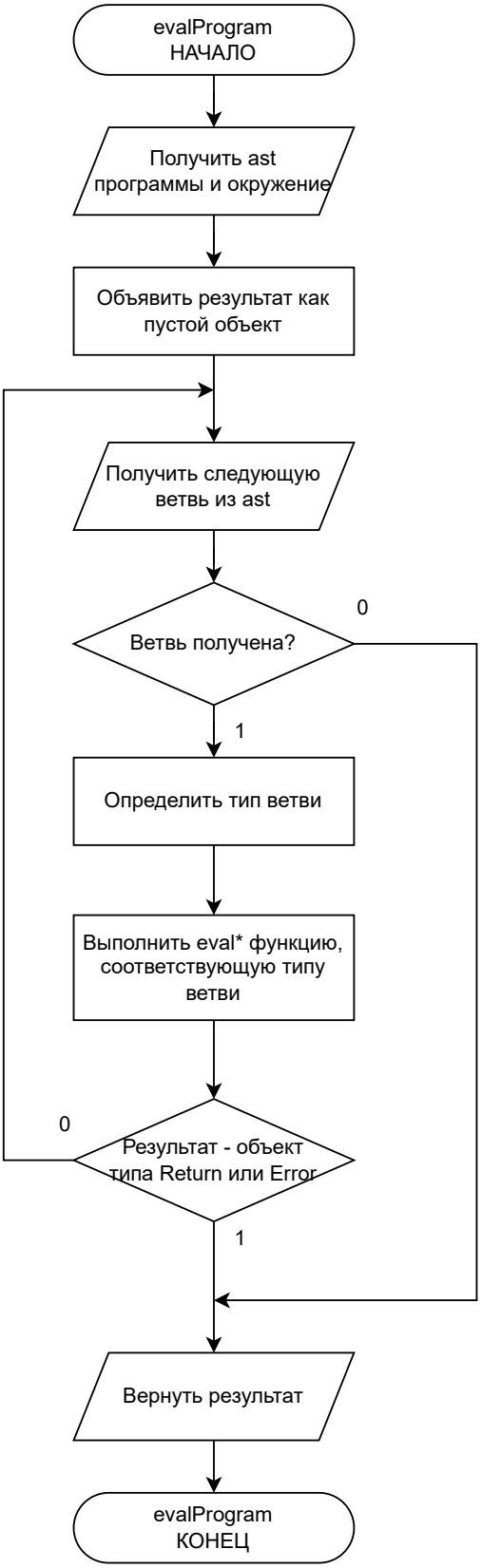

Рисунок 15 – Схема алгоритма «evalProgram»

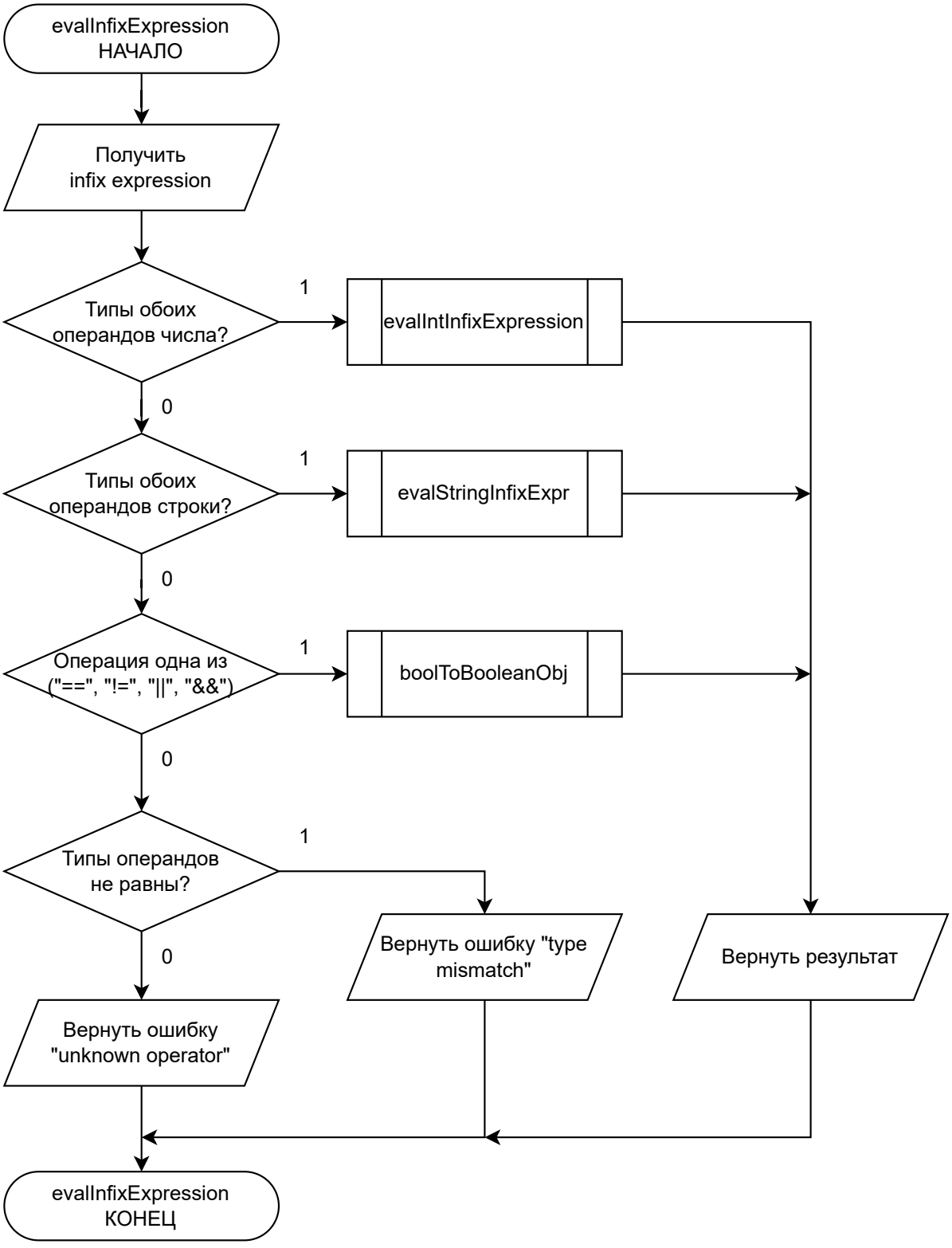

Рисунок 16 – Схема алгоритма «evalInfixExpression»

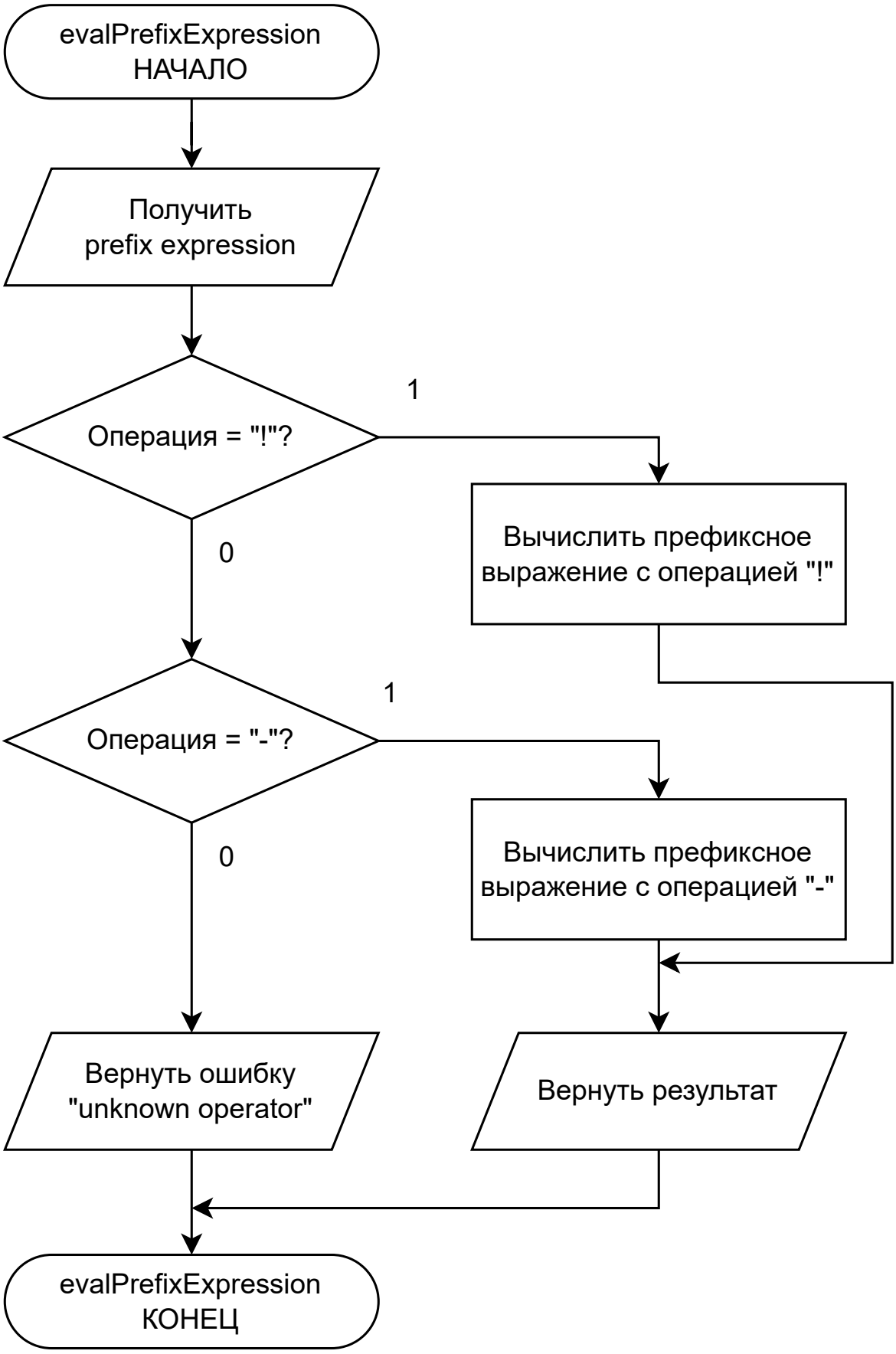

Рисунок 17 – Схема алгоритма «evalPrefixExpression»

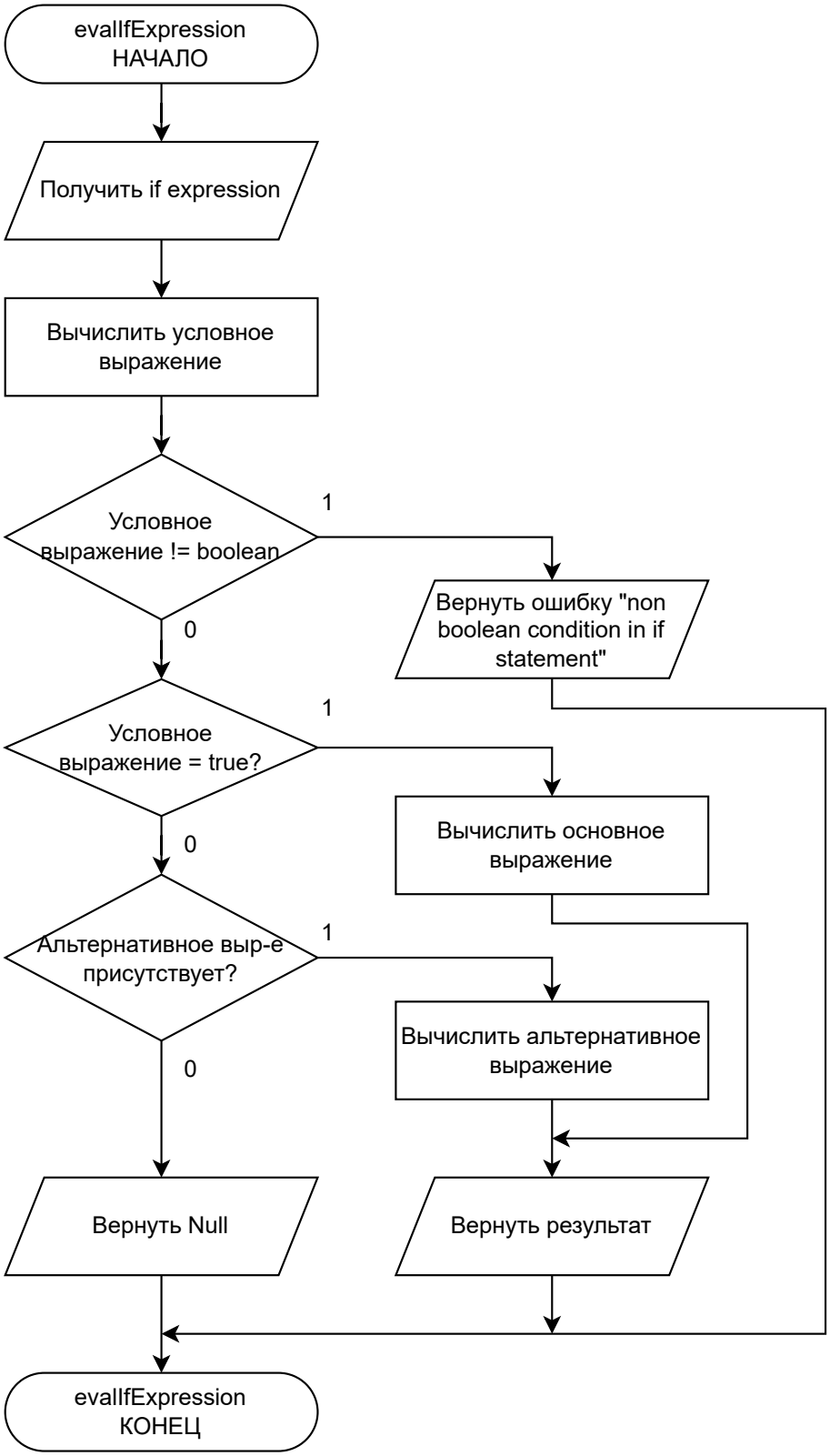

Рисунок 18 – Схема алгоритма «evalIfExpression»

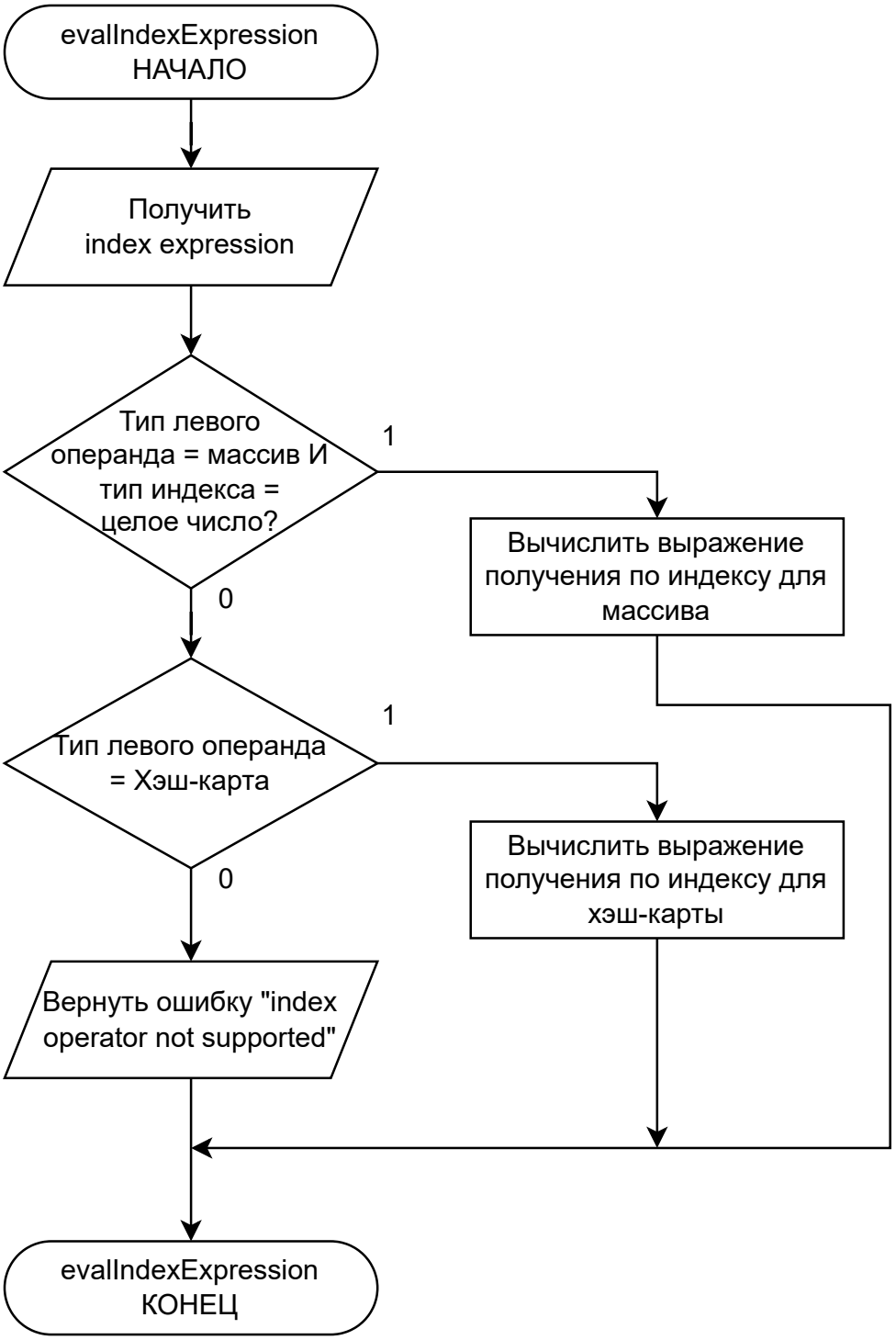

Рисунок 19 – Схема алгоритма «evalIndexExpression»

2.5 Разработка исполнителя

Процесс вычисления выполняется над выражениями из узлов абстрактного синтаксического дерева. Такой вид интерпретатора, который работает с AST называется «tree walking interpreter» или древовидный интерпретатор. В таком интерпретаторе выполняется обход AST и выполнение соответствующих операций для каждого узла.

Последним этапом в процессе обработки исходного кода является его исполнение. До этого шага все выражения языка представляют собой набор символов, токенов или ветви абстрактного синтаксического дерева без какого-либо семантического значения. На данном этапе выражения языка приобретают смысл, то есть начинают интерпретироваться и действовать в соответствии с правилами и инструкциями языка.

Этап исполнения выполняется непосредственно после семантического анализа. Если во время семантического анализа в обрабатываемом выражении не было обнаружено ошибок, выполняется переход к его вычислению. Вычисление выражения выполняется в зависимости от его типа, например, для инфиксного выражения применяется указанная в нем операция над левым и правым операндами и формируется результат в виде объекта соответствующего типа, содержащего результат операции, а для условного выражения сначала вычисляется значение условия, а затем в зависимости от его результата выполняется либо основная ветвь (при значении условия «true»), либо альтернативная (при значении условия «false»).

Только лишь вычислять значения выражений недостаточно. Нужно также сохранять значения переменных для того, чтоб к ним можно было обратиться при обнаружении в выражениях. Чтобы обеспечить эту возможность введем окружение – структуру данных, хранящую информацию о переменных и связанных с ними значениях на время выполнения программы. Таким образом, при объявлении переменной, информация о ней будет записываться в окружение, а при необходимости получить значение этой переменой – ее значение будет считано из окружения.

В качестве примера работы алгоритма на рисунках 20 - 22 приведены схемы алгоритма вычисления некоторых выражений.

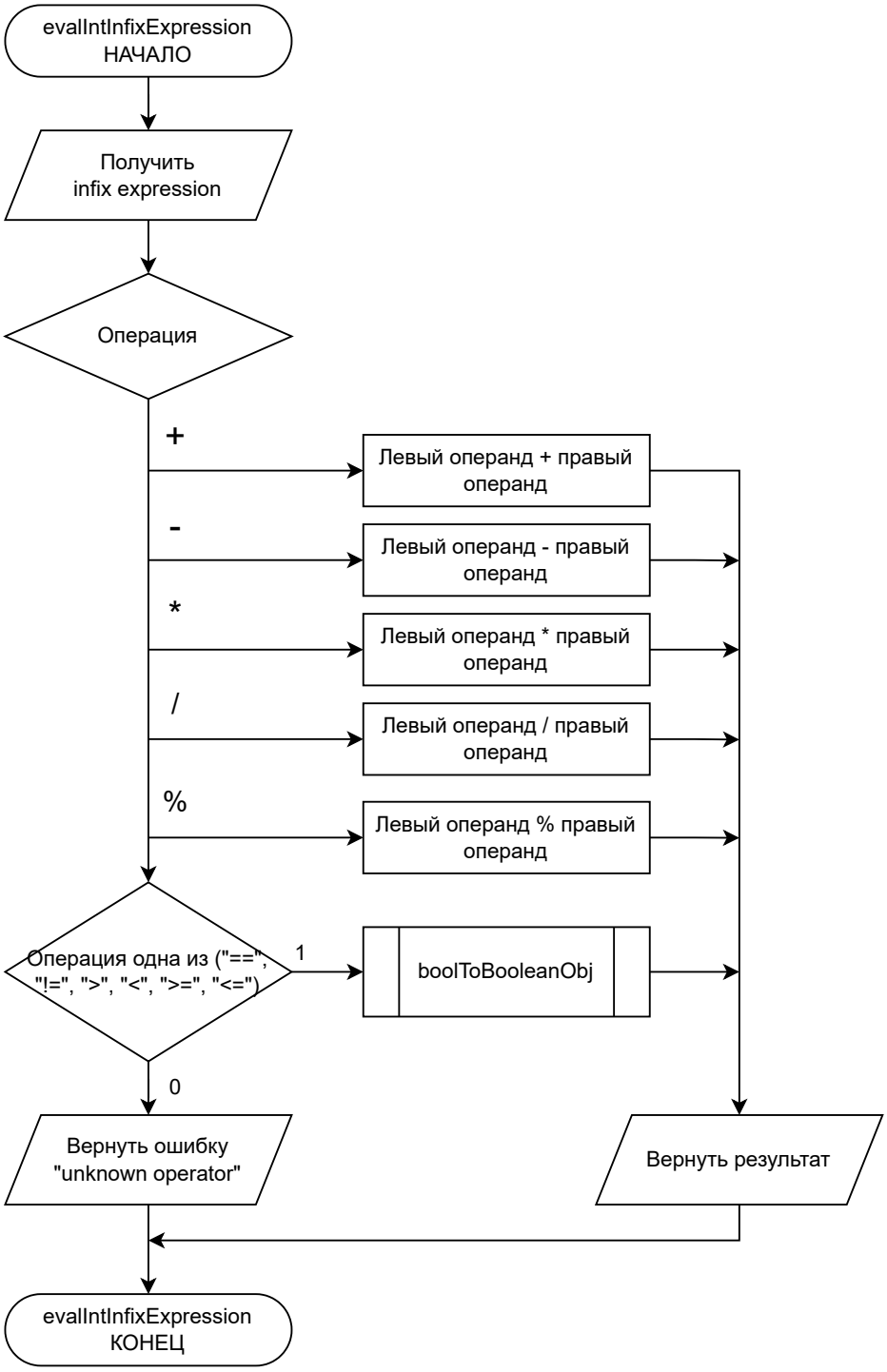

Рисунок 20 – Схема алгоритма вычисления целочисленного инфиксного выражения

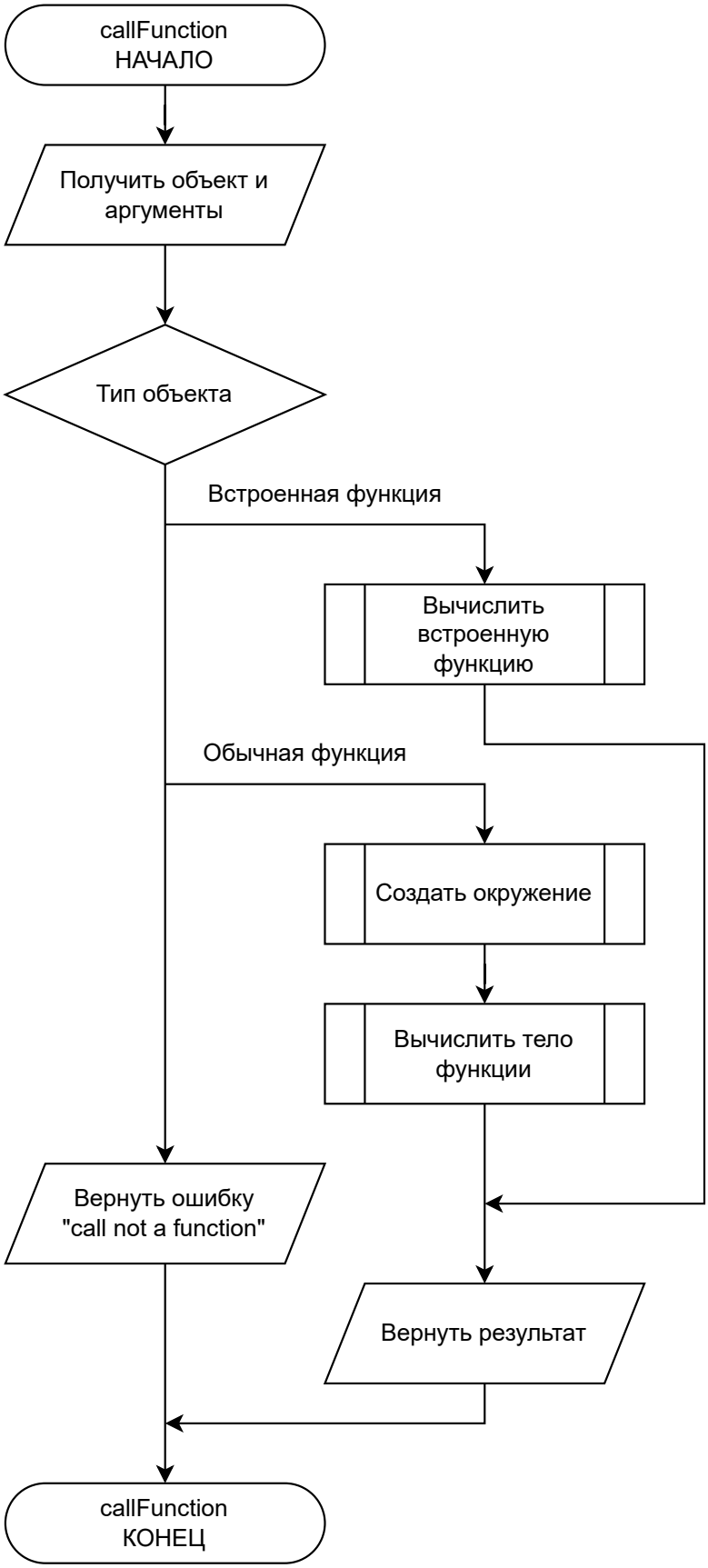

Рисунок 21 – Схема алгоритма исполнения вызова функции

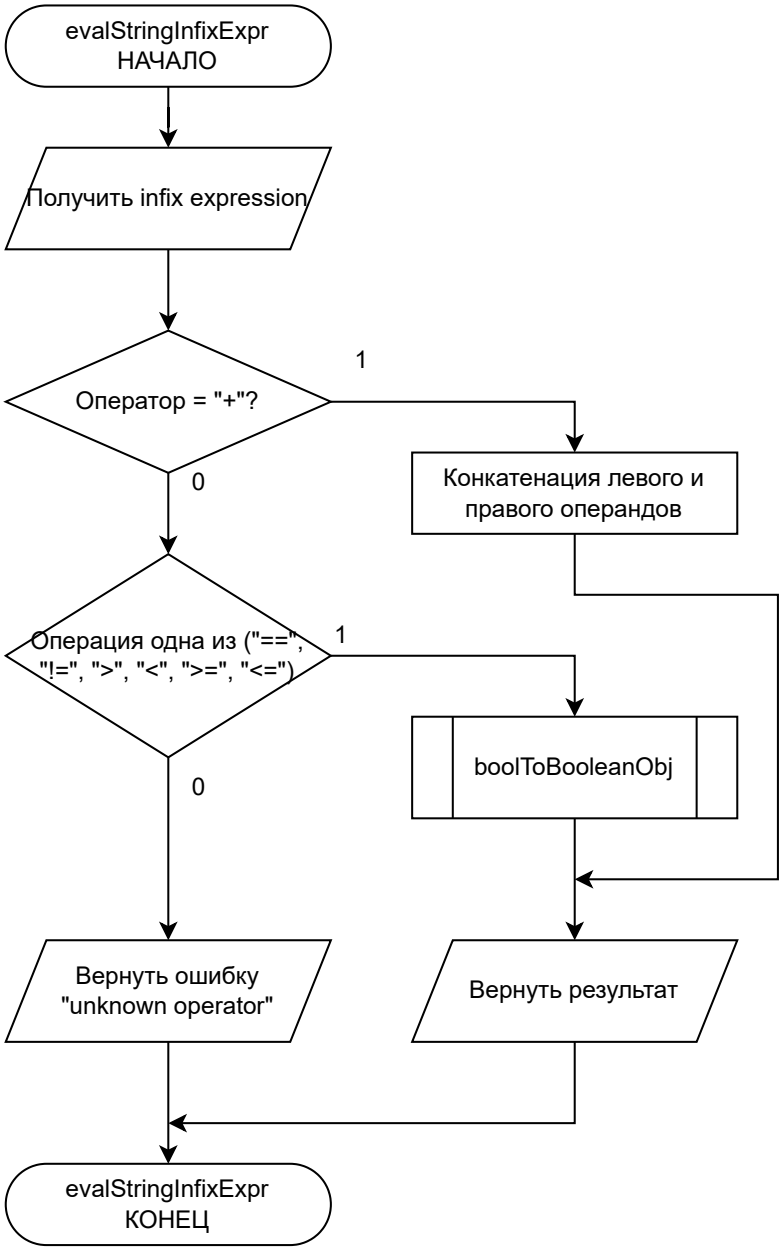

Рисунок 22 – Схема алгоритма вычисления инфиксного выражения для строк

Вывод

В данном разделе были разработаны архитектурно-структурные решения по поставленной задаче. Рассмотрена модульная структура серверной части конструктора, определена связь между модулем компонентов и модулем интерпретации предметно-ориентированного языка.

Рассмотрена обобщенная структура интерпретатора и этапы его работы.

Выполнен обзор способов задания языков, разработана и описана с помощью расширенной формы Бэкуса-Наура формальная грамматика предметно-ориентированного языка. Выполнена разработка структурных решений и алгоритмов функционирования каждого из этапов интерпретатора, а именно: лексического анализатора, синтаксического анализатора, семантического анализатора и исполнителя.

3 Программная реализация

В данном разделе представлено описание выбранных инструментов разработки для решения поставленной задачи. Описана программная реализация модуля интерпретации предметно-ориентированного языка. Рассмотрены детали реализации его основных структурных частей. Также выполнено тестирование основных функций программы.

3.1 Выбор инструментов разработки

В качестве языка программирования для разработки был выбран Go.

Go (Golang) – это компилируемый многопоточный язык программирования с открытым исходным кодом, разработанный компанией Google [9].

Выбор данного языка обусловлен совместимостью между разрабатываемым модулем и основной системой, так как для реализации серверной части конструктора Telegram ботов так же был выбран язык Go. Использование одного языка для всей системы гарантирует, что модули будут легко интегрироваться друг с другом без необходимости создания сложных интерфейсов или адаптеров.

Язык Go имеет и другие положительные характеристики:

– высокая производительность – Go компилируется в машинный код, что обеспечивает его высокую скорость исполнения;

– безопасность типов – строгая статическая типизация предотвращает множество ошибок на этапе компиляции;

– встроенная поддержка параллелизма – в Go реализованы горутины и каналы – встроенные механизмы для эффективной работы с параллельными задачами;

– расширяемость – язык имеет встроенную систему управления модулями и широкую экосистему общедоступных библиотек.

Выбор языка программирования Go для реализации модуля интерпретации предметно-ориентированного языка конструктора Telegram ботов обоснован тем, что он позволяет поддерживать единообразие кода, использовать его технические

преимущества и соответствовать основным требованиям проекта.

3.2 Реализация лексического анализатора

Лексический анализатор состоит из следующих основных компонентов:

– токены – определение структуры и типов токенов, представляющих основные элементы языка;

– конечный автомат – принимает на вход поток символов и определяет токены переходя между состояниями в соответствии с правилами языка.

Токен представляет собой структуру, содержащую информацию о типе токена и его значение, представленное в виде строки. Код структуры, представляющей токен приведен на рисунке 23.

```
type TokenType = string
type Token struct {
    Type    TokenType
    Literal string
}
```

Рисунок 23 – Структура, представляющая токен

Список возможных токенов рассмотрен ранее, см. таблицу 1.

Программную реализацию токенов можно выполнить с помощью списка константных значений. Фрагмент кода реализации токенов представлен на рисунке 24.

Лексер представляет собой структуру, содержащую информацию о входной строке кода, текущем считанном символе, позиции курсора и других технических значениях, необходимых для корректной работы анализатора. Структура представляющая лексер приведена на рисунке 25.

Основная функция лексического анализатора может быть реализована в формате конечного автомата. При получении очередного символа из входной строки кода его необходимо сопоставить с одним из токенов. Стоит заметить, что некоторые токены формируются за счет двух и более символом, например токен

```
IDENT  = "IDENT"  // x, t, add
INT    = "INT"    // 123
STRING = "STRING" // "abcde"
ASSIGN = "="
PLUS   = "+"
STAR   = "*"
LT  = "<"
GT  = ">"
LAND = "&&"
LOR  = "||"

// keywords
IF     = "IF"
ELSE   = "ELSE"
TRUE   = "TRUE"
```

Рисунок 24 – Фрагмент кода реализации токенов

```
type Lexer struct {
    input   string
    ch      byte // current char
    pos     int  // current position (on current char)
    readPos int  // position after current char
    nlsemi  bool // if "true" '\n' translate to ';'
    loPos   token.Pos
}
```

Рисунок 25 – Структура, представляющая лексер

«LAND» (&&), идентификаторы, ключевые слова и т.д. В этом случае, необходимо продолжать получение символов из входной строки до тех пор, пока не будет однозначно определен токен.

Основная функция определения токена выполнена в виде конструкции switch-case. Фрагмент кода представлен на рисунке 26.

```
func (l *Lexer) NextToken() (token.Token, token.Pos) {
    l.skipWhitespace()
    nlsemi := false
    var tok token.Token
    switch l.ch {
    case '\n':
        tok = newToken(token.SEMICOLON, l.ch)
    case '=':
        if l.peekChar() == '=' {
            l.readChar()
            literal := "=="
            tok = token.Token{Type: token.EQ, Literal:
   literal}
        } else {
            tok = newToken(token.ASSIGN, l.ch)
        }
    case '+':
        tok = newToken(token.PLUS, l.ch)
    case '-':
        tok = newToken(token.MINUS, l.ch)
    case '*':
        tok = newToken(token.STAR, l.ch)
    case '/':
        tok = newToken(token.SLASH, l.ch)
    case '!':
        if l.peekChar() == '=' {
            l.readChar()
            literal := "!="
            tok = token.Token{Type: token.NEQ, Literal:
   literal}
        } else {
            tok = newToken(token.EXCLAMINATION, l.ch)
        }
    case '%':
                tok = newToken(token.PERCENT, l.ch)
        case '<':
                if l.peekChar() == '=' {
                        l.readChar()
                        literal := "<="
                        tok = token.Token{Type: token.LEQ,
    Literal: literal}
                } else {
                        tok = newToken(token.LT, l.ch)
                }
```

Рисунок 26 – Фрагмент кода лексера

3.3 Реализация синтаксического анализатора

Разработка синтаксического анализатора включает в себя программную реализацию парсера, способного анализировать токены, получаемые от лексического анализатора и строить абстрактное синтаксическое дерево.

Абстрактное синтаксическое дерево представляет собой структуру, отражающую синтаксическую структуру программы. Узлы AST могут быть двух типов: statement – инструкции и expression – выражения. В соответствии с этим, их программная реализация выполнена в виде интерфейсов, представленных на рисунке 27.

```
type Node interface {
    TokenLiteral() string
    ToString() string
}

// All statement nodes implement
type Statement interface {
    Node
    statementNode()
}

// All expression nodes implement
type Expression interface {
    Node
    expressionNode()
}
```

Рисунок 27 – Интерфейсы узлов AST

Узлы дерева состоят из интерфейса Node. Однако сам по себе он не используется в AST, а необходим для расширения двух вспомогательных интерфейсов Statement и Expression, которые определяют узлы двух типов: инструкции и выражения соответственно.

Пример кода структуры AssignStatement, реализующей интерфейс Statement на рисунке 28.

```
type AssignStatement struct {
    Name  *Ident
    Value Expression
}

func (as *AssignStatement) statementNode()       {}
func (as *AssignStatement) TokenLiteral() string { return
   "" }
func (as *AssignStatement) ToString() string {
    var out bytes.Buffer

    out.WriteString(as.Name.TokenLiteral())
    out.WriteString(" = ")

    if as.Value != nil {
        out.WriteString(as.Value.ToString())
    }

    out.WriteString(";")

    return out.String()
}
```

Рисунок 28 – Пример реализации интерфейса Statement

Пример кода структуры IntegerLineral, реализующей интерфейс Expression представлен на рисунке 29.

```
type IntegerLiteral struct {
    Token token.Token // 5 6
    Value int64
}

func (il *IntegerLiteral) expressionNode()      {}
func (il *IntegerLiteral) TokenLiteral() string { return
   il.Token.Literal }
func (il *IntegerLiteral) ToString() string     { return
   il.Token.Literal }
```

Рисунок 29 – Пример реализации интерфейса Expression

Рассмотрим пример работы синтаксического анализатора.

Входная строка: $5 + 1 * 2/(4 + 9)$.

Результат работы синтаксического анализатора в виде строки с исходным кодом программы, в котором с помощью скобок обозначены приоритеты операторов представлена на рисунке 30.

```
app/app.go:45   (5 + ((1 * 2) / (4 + 9)))
```

Рисунок 30 – Результат работы синтаксического анализатора в виде строки

Графическое представление AST для указанных входных данных представлено на рисунке 31. Дерево, сформированное в результате работы программы для указанных входных данных представлен на рисунке 32.

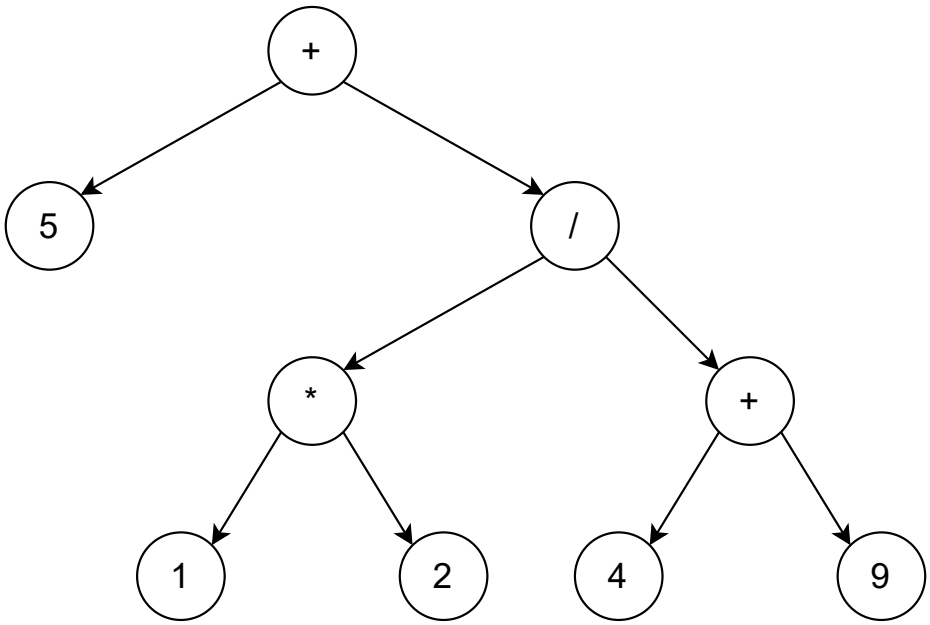

Рисунок 31 – AST для указанной входной строки

```
(*ast.ExpressionStatement)({
|   Token: (token.Token) {
|   |   Type: (string) (len=3) "INT",
|   |   Literal: (string) (len=1) "5"
|   },
|   Expression: (*ast.InfixExpression)({
|   |   Token: (token.Token) {
|   |   |   Type: (string) (len=1) "+",
|   |   |   Literal: (string) (len=1) "+"
|   |   },
|   |   Left: (*ast.IntegerLiteral)({
|   |   |   Token: (token.Token) {
|   |   |   |   Type: (string) (len=3) "INT",
|   |   |   |   Literal: (string) (len=1) "5"
|   |   |   },
|   |   |   Value: (int64) 5
|   |   }),
|   |   Operator: (string) (len=1) "+",
|   |   Right: (*ast.InfixExpression)({
|   |   |   Token: (token.Token) {
|   |   |   |   Type: (string) (len=1) "/",
|   |   |   |   Literal: (string) (len=1) "/"
|   |   |   },
|   |   |   Left: (*ast.InfixExpression)({
|   |   |   |   Token: (token.Token) {
|   |   |   |   |   Type: (string) (len=1) "*",
|   |   |   |   |   Literal: (string) (len=1) "*"
|   |   |   |   },
|   |   |   |   Left: (*ast.IntegerLiteral)({
|   |   |   |   |   Token: (token.Token) {
|   |   |   |   |   |   Type: (string) (len=3) "INT",
|   |   |   |   |   |   Literal: (string) (len=1) "1"
|   |   |   |   |   },
|   |   |   |   |   Value: (int64) 1
|   |   |   |   }),
|   |   |   |   Operator: (string) (len=1) "*",
|   |   |   |   Right: (*ast.IntegerLiteral)({
|   |   |   |   |   Token: (token.Token) {
|   |   |   |   |   |   Type: (string) (len=3) "INT",
|   |   |   |   |   |   Literal: (string) (len=1) "2"
|   |   |   |   |   },
|   |   |   |   |   Value: (int64) 2
|   |   |   |   })
|   |   |   }),
|   |   |   Operator: (string) (len=1) "/",
|   |   |   Right: (*ast.InfixExpression)({
|   |   |   |   Token: (token.Token) {
|   |   |   |   |   Type: (string) (len=1) "+",
|   |   |   |   |   Literal: (string) (len=1) "+"
|   |   |   |   },
|   |   |   |   Left: (*ast.IntegerLiteral)({
|   |   |   |   |   Token: (token.Token) {
|   |   |   |   |   |   Type: (string) (len=3) "INT",
|   |   |   |   |   |   Literal: (string) (len=1) "4"
|   |   |   |   |   },
|   |   |   |   |   Value: (int64) 4
|   |   |   |   }),
|   |   |   |   Operator: (string) (len=1) "+",
|   |   |   |   Right: (*ast.IntegerLiteral)({
|   |   |   |   |   Token: (token.Token) {
|   |   |   |   |   |   Type: (string) (len=3) "INT",
|   |   |   |   |   |   Literal: (string) (len=1) "9"
|   |   |   |   |   },
|   |   |   |   |   Value: (int64) 9
|   |   |   |   })
|   |   |   })
|   |   })
|   })
})
```

Рисунок 32 – Результат работы парсера в виде AST

3.4 Реализация семантического анализатора

Прежде, чем переходить непосредственно к реализации семантического анализатора, необходимо реализовать объектную систему. Объектная система является основополагающей частью семантического анализатора и исполнителя. Семантической анализатор в процессе своей работы выполняет необходимые проверки на основе значений, представленных в виде объектов внутреннего представления.

Каждое значение выражения представляется в виде структуры, которая соответствует некоторому объекту интерфейсного типа – рисунок 33.

```
type ObjectType string

type Object interface {
    Type() ObjectType
    ToString() string
}
```

Рисунок 33 – Интерфейс объекта

На рисунке 34 приведен пример структуры, которая представляет данные типа «целое число».

```
func (i *Integer) Type() ObjectType { return INTEGER_OBJ }
func (i *Integer) ToString() string {
    return fmt.Sprintf("%d", i.Value)
}
func (i *Integer) HashKey() HashKey {
    return HashKey{Type: i.Type(), Value: uint64(i.Value)}
}
```

Рисунок 34 – Пример реализации интерфейса Object

Поле «Value» предназначено для хранения значения числа. Метод «Type()» возвращает информацию о принадлежности структуры типу «integer». Метод

«ToString()» формирует хранимое значение в виде строки. Используется для читаемого представления значения объекта в процессе отладки.

Подобные структуры, реализующие интерфейс «Object» представлены для всех примитивных и составных типов данных, используемых в языке: boolean, string, integer, array, HashMap. Кроме этого, реализованы еще несколько вспомогательных структур:

– «Null» для поддержки соответствующих значений;

– «Error», содержащая информацию об ошибке, возникшей на этапе семантического анализа;

– «Return» для представления возвращаемых значений;

– «Function» - специальная структура, используемая при обработке вызова функции;

– «Builtin» - структура, представляющая встроенные функции.

Данные объекты формируют объектную систему внутреннего представления значений программы.

Семантический анализ выполняется во время рекурсивного прохода по узлам AST. В ходе рекурсии при достижении примитивных значений в крайних узлах ветвей AST для каждого значения создается объект внутреннего представления соответствующего типа. На обратном ходу рекурсии, при необходимости выполнить проверку семантической корректности выражения она выполняется над объектами внутреннего представления, сформированными ранее. При анализе генерируется ошибка, в случае её обнаружения, в противном случае начинается вычисление значение выражения исполнителем.

Фрагмент кода функции разбора инфиксного выражения представлен на рисунке 35.

```
func evalInfixExpression(op string, left object.Object, right
    object.Object) object.Object {
    switch {
    case left.Type() != right.Type():
        return newError("type mismatch: %s %s %s", left.Type
   (), op, right.Type())
    case left.Type() == object.INTEGER_OBJ && right.Type() ==
    object.INTEGER_OBJ:
        return evalIntInfixExpr(op, left, right)
    case left.Type() == object.STRING_OBJ && right.Type() ==
   object.STRING_OBJ:
        return evalStringInfixExpr(op, left, right)
    case op == "==":
        return boolToBooleanObj(left == right)
    case op == "!=":
        return boolToBooleanObj(left != right)
    case op == "||":
        return boolToBooleanObj(left.(*object.Boolean).Value
   || right.(*object.Boolean).Value)
    case op == "&&":
        return boolToBooleanObj(left.(*object.Boolean).Value
   && right.(*object.Boolean).Value)
    default:
        return newError("unknown operator: %s %s %s", left.
   Type(), op, right.Type())
    }
}
```

Рисунок 35 – Пример семантического анализа инфиксного выражения

3.5 Реализация исполнителя

В общем виде процесс исполнения тесно связан с этапом семантического анализа. Выполняется рекурсивный обход абстрактного синтаксического дерева. Первым шагом каждое выражение проходит семантическую проверку. После успешного завершения семантического анализа выражения из AST передаются на этап их вычисления. Разнотипные выражения обрабатываются по-разному, однако результат вычисления всегда представляет собой некоторый тип данных, представленный в виде объекта – внутреннего представления.

Окружение для хранения информации о переменных в программном коде реализовано в виде структуры, содержащей поля с хэш-картой с самими переменными и их значениями, а также ссылка на эту же структуру для организации области видимости при вызове функций. На рисунке 36 приведен пример программной реализации данной структуры.

```
type Env struct {
    store map[string]Object
    outer *Env
}
```

Рисунок 36 – Реализация окружения

За обработку узлов AST отвечает единственная функция, которая определяет тип узла и передает управление соответствующей функции, которая после прохождения семантической проверки вычисляет значение для узла данного типа. В конечном итоге формируется внутреннее представление в виде объектов с примитивными типами данных, либо объекты представляющие составные типы, состоящие из примитивных, так как массивы и хеш-карты. Фрагмент кода основной функции получения и обработки узлов AST приведен на рисунке 37.

При обнаружении в коде определения переменной, ее необходимо сохранить в памяти, чтобы в дальнейшем иметь к ней доступ. Для этого используется окружение. Переменные в окружении хранятся в виде хэш-карты, ключи которой представляют идентификатор переменной, а значения – внутреннее представление значений переменной, то есть объекты.

Код функций записи переменной в окружение и получения из него приведен на рисунке 38.

```
func Eval(n ast.Node, env *object.Env) object.Object {
    switch node := n.(type) {
    case *ast.Program:
        return evalProgram(node, env)
    case *ast.BlockStatement:
        return evalBlockStatement(node, env)
    case *ast.ExpressionStatement:
        return Eval(node.Expression, env)
    case *ast.ReturnStatement:
        val := Eval(node.Value, env)
        if isError(val) {
            return val
        }

        return &object.Return{Value: val}
    case *ast.AssignStatement:
        val := Eval(node.Value, env)
        if isError(val) {
            return val
        }

        env.Set(node.Name.Value, val)
    case *ast.IntegerLiteral:
        return &object.Integer{Value: node.Value}
    case *ast.Boolean:
        if node.Value {
            return TRUE
        }
        return FALSE
```

Рисунок 37 – Фрагмент кода исполнителя

```
func (e *Env) Get(key string) (Object, bool) {
    obj, ok := e.store[key]
    if !ok && e.outer != nil {
        obj, ok = e.outer.Get(key)
    }
    return obj, ok
}
func (e *Env) Set(key string, val Object) Object {
    e.store[key] = val
    return val
}
```

Рисунок 38 – Код функций записи и получения значений окружения

Использование единого хранилища значений переменных для всей области видимости программы вносит некоторые ограничения. Например в аргументах функции могут быть определены параметры, имена которых совпадают с объявленным ранее переменным. Это некорректное поведение, так как первое объявленное значение будет перезаписано другим, переданным в функцию. Пример кода такой ситуации приведен на рисунке 39.

```
x = 10;
f = func(x) {
    return x * 10;
}
f(5);
x; //5
```

Рисунок 39 – Пример кода некорректного поведения

В данном коде, имя параметра функции совпадает с именем переменной – x. Переменная «x» объявлена со значением, равным 10. Данное значение перезапишется значением, переданным в функцию при ее вызове, в данном случае значением, равным 5. Таким образом, после вызова функции значение переменной «x» будет неявно изменено и станет равным 5. Данная логика работы является ошибочной.

Решение лежит в выделении внутреннего окружения функции при её вызове. Именно для такого случая в ранее рассмотренной структуре Env содержится ссылка на другой экземпляр структуры такого же типа. Так, при вызове функции, необходимо создать новый экземпляр окружения, записать в него переданные аргументы и установить ссылку на внешнее окружение, то из которого была вызвана функция. Такой подход позволит корректно выполнять вложенные функции и рекурсивные вызовы.

Для строк и массивов реализованы несколько встроенных функций:

– len – определение длины строки или массива;
– push – добавление элемента в конец массива;
– first – получение первого элемента массива;
– last – получение последнего элемента массива.

Хэш-карты реализованы на базе хэш-карт языка Go. В качестве ключа могут быть следующие типы данных: строка, булево значение, число. Так как данные типы представлены в виде внутренних объектов, нельзя брать тип Object в качестве ключа. При занесении значения в хэш-карту и попытке последующего его получения ключи будут представлять разные экземпляры несмотря на одинаковое значение. На рисунке 40 приведен наглядный пример.

```
X = { "name": "Bob" }
X["name"]
```

Рисунок 40 – Пример получения значения хэш-карты

Строковой ключ «name» при попытке получить значение из карты не будет возвращать значение ”Bob так как при вычислении выражения будет создан новый экземпляр объекта, представляющего строку. Для решения этой проблемы можно использовать в качестве ключа строку, содержащую хэш от значения объекта.

Фрагмент кода объектной системы с реализацией хэш-карт представлен на рисунке 41.

Фрагмент кода функции вычисления целочисленного инфиксного выражения приведен на рисунке 42.

```
type HashKey struct {
    Type  ObjectType
    Value uint64
}

type Hashable interface {
    HashKey() HashKey
}

type HashPair struct {
    Key   Object
    Value Object
}

type HashMap struct {
    Pairs map[HashKey]HashPair
}

func (h *HashMap) Type() ObjectType { return HASH_MAP_OBJ
   }
func (h *HashMap) ToString() string {
    var out bytes.Buffer

    pairs := []string{}
    for _, el := range h.Pairs {
        pairs = append(pairs, fmt.Sprintf("%s: %s", el.Key
   .ToString(), el.Value.ToString()))
    }

    out.WriteString("[")
    out.WriteString(strings.Join(pairs, ", "))
    out.WriteString("]")

    return out.String()
}
```

Рисунок 41 – Фрагмент кода реализации хэш-карт

```
func evalIntInfixExpr(op string, left object.Object, right
    object.Object) object.Object {
    lVal := left.(*object.Integer).Value
    rVal := right.(*object.Integer).Value
    switch op {
    case "+":
        return &object.Integer{Value: lVal + rVal}
    case "-":
        return &object.Integer{Value: lVal - rVal}
    case "*":
        return &object.Integer{Value: lVal * rVal}
    case "/":
        return &object.Integer{Value: lVal / rVal}
    case "%":
        return &object.Integer{Value: lVal % rVal}
    case "==":
        return boolToBooleanObj(lVal == rVal)
    case "!=":
        return boolToBooleanObj(lVal != rVal)
    case "<":
        return boolToBooleanObj(lVal < rVal)
    case ">":
        return boolToBooleanObj(lVal > rVal)
    case "<=":
        return boolToBooleanObj(lVal <= rVal)
    case ">=":
        return boolToBooleanObj(lVal >= rVal)
    default:
        return newError("unknown operator: %s %s %s", left
   .Type(), op, right.Type())
    }
}
```

Рисунок 42 – Фрагмент кода функции вычисления целочисленного инфиксного выражения

Пример успешного выполнения программы для указанных входных данных представлен на рисунке 43.

Входная строка: $5 + 1 * 20/(5 + 5)$.

```
app/app.go:39   7
```

Рисунок 43 – Пример успешного выполнения программы

Рассмотрим пример завершения программы с семантической ошибкой. Исходный код с такой ошибкой представлен на рисунке 44. Результат выполнения показан на рисунке 45.

```
x = 5 + 1 * 20 / (5 + 5)
if (x > true) {
    return 1
}
```

Рисунок 44 – Пример кода, содержащего ошибку

```
app/app.go:39   error: type mismatch: INTEGER > BOOLEAN
```

Рисунок 45 – Завершение работы программы с семантической ошибкой

Листинг фрагментов кода реализации основных частей разработанного программного продукта представлен в приложении Б.

Вывод

В данном разделе на основании разработанных архитектурно-структурных решениях и алгоритмах функционирования выполнена программная реализация модуля интерпретации предметно-ориентированного языка. С помощью выбранных инструментов разработки выполнена программная реализация лексического анализатора, синтаксического анализатора, семантического анализатора и исполнителя.

4 Тестирование и экспериментальная апробация

В данном разделе представлены этапы тестирования и экспериментальной апробации разработанного программного продукта.

4.1 Тестирование

Тестирование является одним из ключевых этапов разработки, направленным на подтверждение корректности работы программы.

Выделяют три основных вида тестирования [10]:

– модульное тестирование;
– интеграционное тестирование;
– функциональное тестирование.

Модульное тестирование или юнит-тестирование позволяет проверить отдельные изолированные компоненты системы. Оно направлено на проверку правильности функционирования каждого блока с помощью входных данных и подтверждения соответствия результата теста ожидаемым результатам.

Интеграционное тестирование направлено на проверку корректности взаимодействия нескольких модулей как единой группы. Интеграционные тесты позволяют выявить проблемы, связанные с зависимостями и обменом информацией между компонентами системы.

Функциональное тестирование – тип тестирования, который направлен на проверку соответствия работы программы заданной функциональности. Основная цель – убедиться, что разработанное программное обеспечение работает правильно и обеспечивает требуемую функциональность.

Проверка корректности работы лексера, парсера, семантического анализатора и исполнителя выполнена с помощью модульных тестов. Написаны тесты для большинства основных функций. Некоторые тест-кейсы приведены в таблицах 3-13.

Для реализации и запуска тестов в разработанной программе использовалась встроенная среду выполнения Go утилита «go test». Эта утилита предостав-

ляет удобный и мощный инструмент для создания и выполнения тестов, включая модульные и функциональные тесты. «Go test» позволяет одной командой запускать все реализованные тест-кейсы в проекте и получать результат их выполнения [11].

В ходе запуска тестирования все тесты выполнились успешно. Результат тестирования представлен на рисунке 46.

```
go test ./... | { grep -v 'no test files'; true; }
ok      github.com/botscubes/bql/internal/evaluator     0.024s
ok      github.com/botscubes/bql/internal/lexer 0.022s
ok      github.com/botscubes/bql/internal/parser        0.021s
```

Рисунок 46 – Результаты запуска тестов

Таблица 3 – Тест-кейсы исполнения целочисленного выражения

Входные данные	Ожидаемый результат
4	4
-5	-5
2 + 2	4
1 + 2 + 3 + 4 + 5 - 1 - 2 - 3	9
2 * 3 * 4 * 5 * 6 * 7 * 8 * 9	362880
10 + 10 * 2	30
(10 + 10) * 2	40
100 / 2 * 2 + 5	105
100 / (2 * 2) - 200	-175
5 % 2	1
4 % 2	0

Таблица 4 – Тест-кейсы исполнения встроенных функций

Входные данные	Ожидаемый результат
len(”abc”)	3
len(”abc”+ ”efg”)	6
len()	0
len(1)	type of argument not supported: INTEGER
len(”a ”b”)	wrong number of arguments: 2 want: 1
x = ”abc”; len(x)	3
len([])	0
x = [1, 2, 3]; len(x)	3
push([], 4)	[4]
push([1, 2, 3], 4)	[1, 2, 3, 4]
push(”a 4)	first argument must be ARRAY, got: STRING”
first([])	null
first([1])	1
first([3, 2, 1])	3
first(”a”)	argument must be ARRAY, got: STRING
last([])	null
last([1])	1
intToString(”a”)	argument must be INTEGER, got: STRING
intToString(123)	123
intToString(1, 2)	wrong number of arguments: 2 want: 1

Таблица 5 – Тест-кейсы исполнения условного выражения

Входные данные	Ожидаемый результат
if (true) { 50 }	50
if (false) { 50 }	null
if (!false) { 50 }	50
if (1 <2) { 50 } else { 100 }	50
if (1 >2) { 50 } else { 100 }	100
if (true || false) { 50 } else { 100 }	50
if (true && false) { 50 } else { 100 }	100

Таблица 6 – Тест-кейсы семантических ошибок

Входные данные	Ожидаемый результат
true + false	unknown operator: BOOLEAN + BOOLEAN
1; true - false; 2	unknown operator: BOOLEAN - BOOLEAN
1; true + false + true + true; 2	unknown operator: BOOLEAN + BOOLEAN
-true	unknown operator: -BOOLEAN
true + 3	type mismatch: BOOLEAN + INTEGER
3 * false	type mismatch: INTEGER * BOOLEAN
”Hello”* 3	type mismatch: STRING * INTEGER
”Hello”* ”Earth”	unknown operator: STRING * STRING
if (3) { 1 }	non boolean condition in if statement
x = 10; q	identifier not found: q
true[1]	index operator not supported: BOOLEAN
123[123]	index operator not supported: INTEGER

Таблица 7 – Тест-кейсы исполнения вызова функции

Входные данные	Ожидаемый результат
x = fn(x){ x }; x(10);	10
x = fn(x, y){ return x * y }; x(10, 9);	90
x = fn(x, y, z){ return x * (y - z) }; x(10, 9, 1);	80
x = fn(x, y){ return x + y }; x(10, x(x(1, 1), x(3, 5)));	20
fn(x){ x }(5)	5

c = fn(x){
fn(y) { x + y }
}
a = c(5);
a(4)
	9

Таблица 8 – Тест-кейсы исполнения массива

Входные данные	Ожидаемый результат
[1, 2, -33, 5+5, 1 + 2 + 3 + 4 * 5]	[1, 2, -33, 10, 26]

Таблица 9 – Тест-кейсы исполнения строкового выражения

Входные данные	Ожидаемый результат
”Hello Earth”	Hello Earth
”Hello”+ + ”Earth”	Hello Earth

Таблица 10 – Тест-кейсы исполнения булева выражения

Входные данные	Ожидаемый результат
true	true
false	false
1 == 1	true
1 != 1	false
1 <2	true
1 >2	false
1 <= 2	true
1 <= 1	true
1 >= 2	false
1 >= 1	true
true == true	true
false == false	true
true == false	false
true != false	true
(true == false) == false	true
(1 == 1) == true	true
(1 <= 1) == false	false
true || false	true
true && false	false
true && true	true
false && false	false
false || false	false
!true	false
!false	true
!!false	false

Таблица 11 – Тест-кейсы исполнения индексного выражения для массива

Входные данные	Ожидаемый результат
[1, 2, 5][0]	1
[1, 2, 5][2]	5
[1, 2, 5][3]	null
[1, 2, 5][-1]	null
[1, 2, 5][1+1]	5
x = 1; [1, 2, 5][x]	2
a = [1, 2, 5]; a[0] + a[1] * a[2]	11

Таблица 12 – Тест-кейсы исполнения индексного выражения для хэш-карты

Входные данные	Ожидаемый результат
{”x”: 1}[”x”]	1
{”x”: 1}[”y”]	null
{}[”y”]	null
{5: 2}[5]	2
{true: 5}[true]	5

4.2 Экспериментальная апробация

Экспериментальная апробация направлена на демонстрацию возможностей разработанного предметно-ориентированного языка.

Рассмотрим пример решения некоторой задачи с помощью DSL. Пусть необходимо реализовать простую корзину товаров с возможностью многоразового выбора, просмотра списка выбранных товаров и их суммарной стоимости.

Полная сконструированная структура бота представлена на рисунке 47. Далее рассмотрим детально этапы построения данной структуры бота.

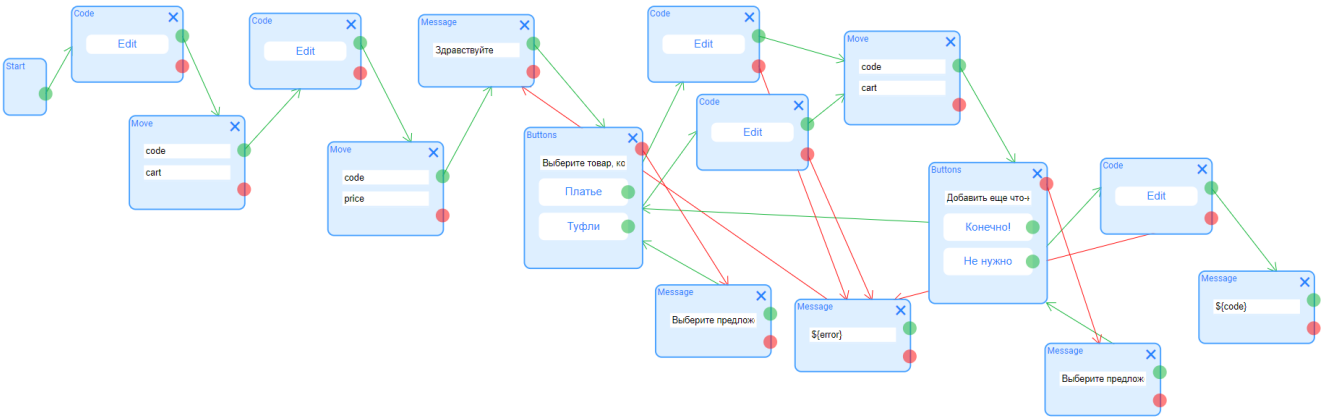

Рисунок 47 – Полная компонентная структура бота

Первоначально необходимо построить в визуальном конструкторе цепочку блоков, определяющую пользовательскую корзину и стоимость товара. Для простоты список товаров будет состоять из двух позиций. Цепочка, определяющая корзину и стоимость товара представлена на рисунке 48. Код с объявлением стоимости единицы товара приведен на рисунке 49.

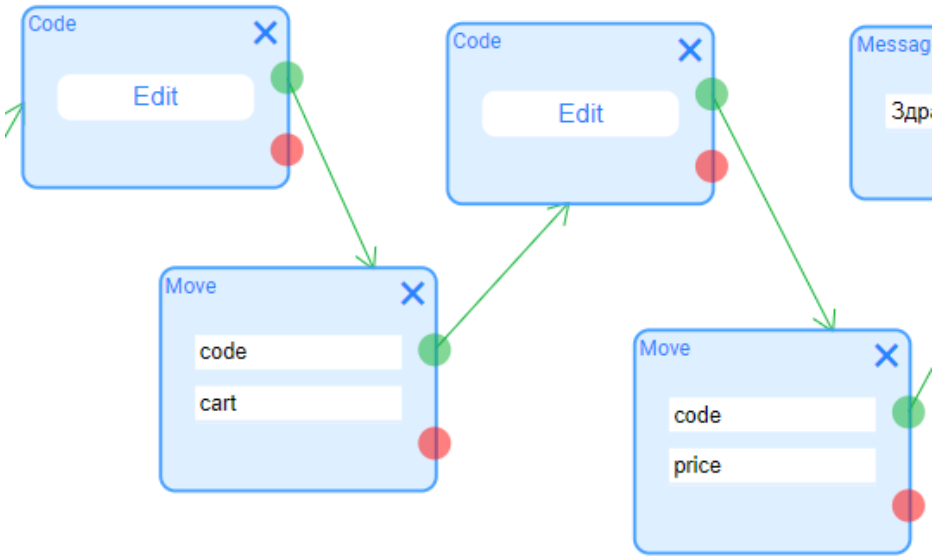

Рисунок 48 – Цепочка, определяющая корзину и стоимость товара

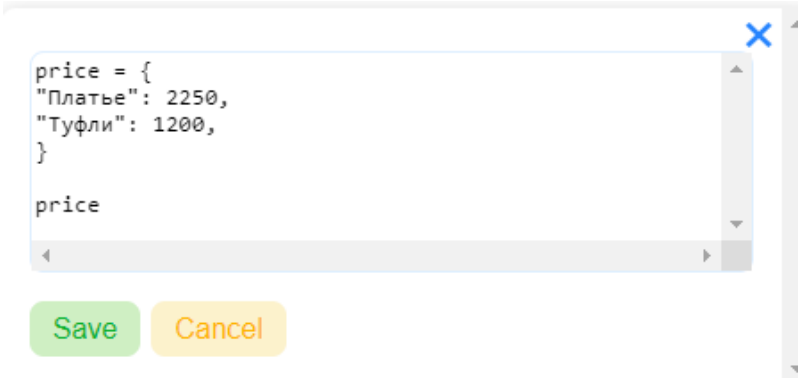

Рисунок 49 – Пример кода на DSL

Затем необходимо предоставить пользователю бота выбор товара. Для этого отправим сообщение с двумя кнопками, обозначающими возможные варианты ответа. Если ответ пользователя не будет принадлежать этим возможным вариантам, запрос будет отправлен повторно. После выбора пользователем определенного товара, увеличим количество данного товара на 1 с помощью компонента «код». Цепочка выбора товара представлена на рисунке 50.

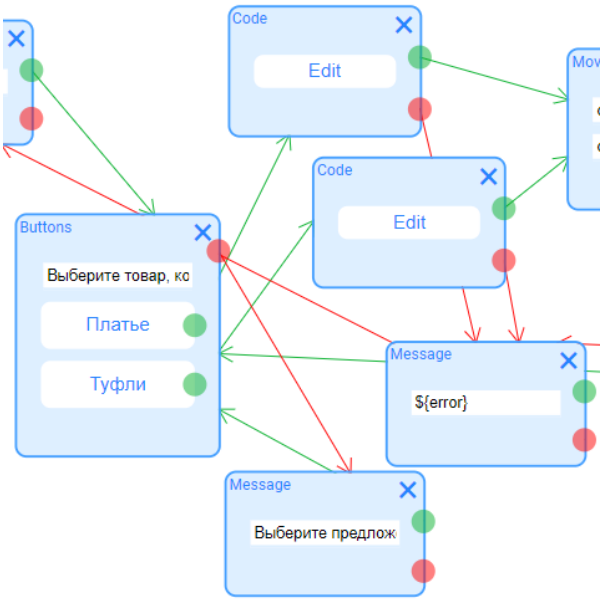

Рисунок 50 – Цепочка выбора товара

Код увеличения количества единиц товара в корзине при его выборе пользователем представлен на рисунке 51.

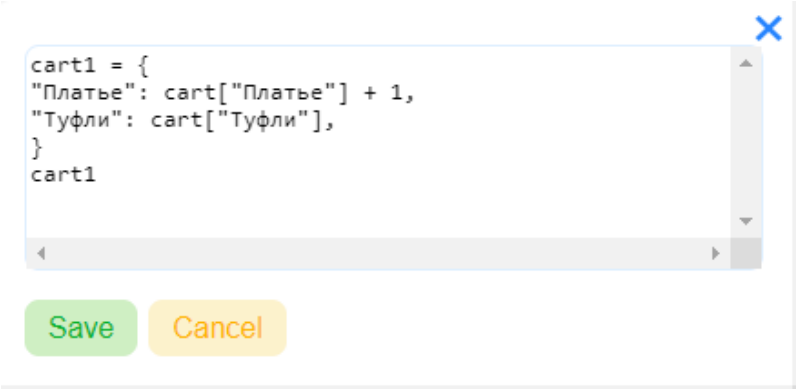

Рисунок 51 – Код добавления товара в корзину

После того, как пользователь выбрал товар, выполняется отправка сообщения с предложением повторного выбора товара. В случае отказа выполняется формирование списка товаров из корзины с указанием их стоимости и общей суммы по всем позициям корзины. Пример кода на предметно-ориентированном языке, который формирует ответ приведен на рисунке 52.

```
pprice = cart["Платье"] * price["Платье"];
tprice = cart["Туфли"] * price["Туфли"];

t = ""
if(cart["Платье"] > 0) {
        t = t + "- Платье: " + intToString(cart["Платье"])
    + " ед.: " + intToString(pprice) + " р. \n"
}

if(cart["Туфли"] > 0) {
        t = t + "- Туфли: " +  intToString(cart["Туфли"])
   + " ед.: " + intToString(tprice) + " р. \n"
}

if(cart["Туфли"] == 0 && cart["Платье"] == 0) {
        t = "Корзина пуста\n"
}

s = "Ваша корзина:\n " + t + "\n Общая стоимость: " +
   intToString(pprice + tprice) + " р."
s
```

Рисунок 52 – Код формирования сообщения с корзиной

Демонстрация работы запущенного Telegram бота представлена на рисунке 53.

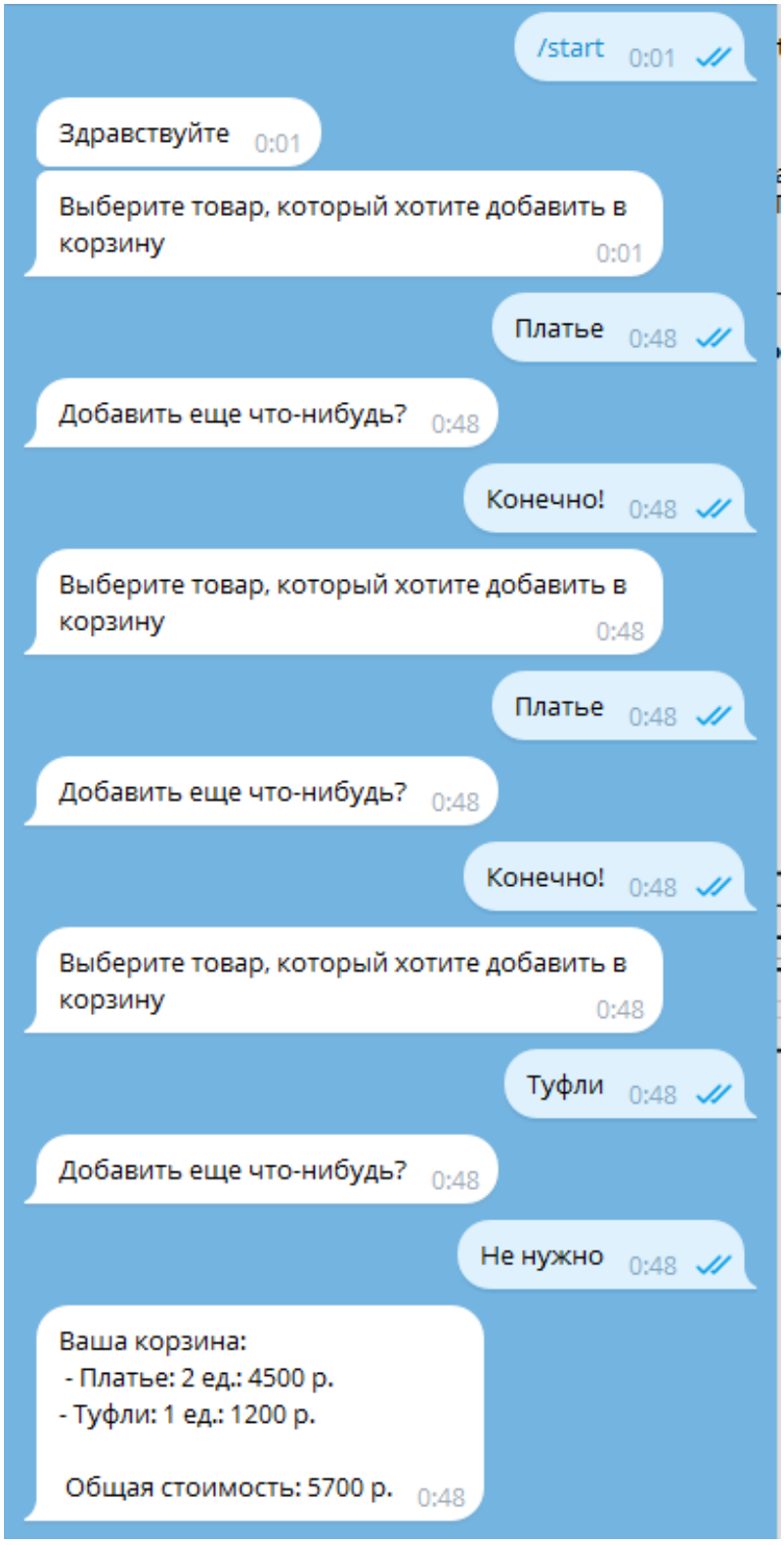

Рисунок 53 – Демонстрация работы бота

Вывод

В данном разделе было проведено тестирование разработанного программного продукта. На базе встроенной в среду выполнения Go утилиты «go test» были реализованы юнит-тесты, охватывающие большинство функций модуля интерпретации.

Также была выполнена экспериментальная апробация, показывающая пример расширения возможностей визуального конструктора Telegram ботов за счет использования компонента, для запуска кода на разработанном предметно-ориентированном языке.

Заключение

В ходе выполнения выпускной квалификационной работы была проведена разработка предметно-ориентированного языка для конструктора Telegram ботов и интерпретатора для него. Основной целью проекта было расширение функциональных возможностей визуального конструктора Telegram ботов. Благодаря использованию DSL можно более гибко настроить логику работы бота и предоставить пользователю платформы возможность разработки сложных, уникальных ботов, которых было бы невозможно или затруднительно построить с использованием только базового набора компонентов визуального конструктора.

На основе анализа предметной области и определения основных функциональных возможностей предметно-ориентированного языка было сформировано расширенное технического задание, определяющее требования к разрабатываемому продукту.

Исходя из требований технического задания первоочередной задачей было разработать и описать грамматику предметно-ориентированного языка. Для описания формальной грамматики была выбрана расширенная форма Бэкуса-Наура.

Была выполнена разработка архитектурно-структурных решений и алгоритмов функционирования каждой составляющей модуля интерпретации, а именно лексического, синтаксического, семантического анализаторов и исполнителя инструкций предметно-ориентированного языка. Также описан программный интерфейс интеграции модуля с серверной частью конструктора.

На этапе программной реализации были выполнено кодирование анализаторов и исполнителя с помощью выбранных инструментов разработки в соответствии со структурными решениями и алгоритмами функционирования.

Кроме этого было осуществлено модульное тестирование разработанной программы, в ходе которого все тесты были выполнены успешно. Также выполнена экспериментальная апробация с примером использования разработанного программного продукта.

Разработанный предметно-ориентированный язык и модуль интерпретации для него решают проблему ограниченных функциональных возможностей кон-

структора Telegram ботов. Теперь пользователь визуального конструктора может реализовывать множество уникальных идей из различных предметных областей, не ограничиваясь стандартным набором компонентов.

Исходный код разработанного модуля интерпретации выложен в публичный доступ на Github под открытой лицензией MIT. Открытый исходный код под данной лицензией предоставляет возможность любому пользователю вносить изменения, разворачивать решение на своей стороне и эксплуатировать в коммерческих целях.

Приложение А

(обязательное)

Авторская справка

Я, Крючков Илья Сергеевич, автор выпускной квалификационной работы «Разработка конструктора Telegram-ботов. Часть 2» сообщаю, что мне известно о персональной ответственности автора за разглашение сведений, подлежащих защите законами РФ о защите объектов интеллектуальной собственности.

Одновременно сообщаю, что:

1. При подготовке к защите выпускной квалификационной работы не использованы источники (документы, отчёты, диссертации, литература и т.п.), имеющие гриф секретности или «Для служебного пользования» ФГБОУ ВО «Вятский государственный университет» или другой организации.

2. Данная работа не связана с незавершёнными исследованиями или уже с завершёнными, но ещё официально не разрешёнными к опубликованию ФГБОУ ВО «Вятский государственный университет» или другими организациями.

3. Данная работа не содержит коммерческую информацию, способную нанести ущерб интеллектуальной собственности ФГБОУ ВО «Вятский государственный университет» или другой организации.

4. Данная работа не является результатом НИР или ОКР, выполняемой по договору с организацией.

5. В предлагаемом к опубликованию тексте нет данных по незащищённым объектам интеллектуальной собственности других авторов.

6. Использование моей дипломной работы в научных исследованиях оформляется в соответствии с законодательством РФ о защите интеллектуальной собственности отдельным договором.

Автор: Крючков И. С. «____» _____ 2024 г. _____

подпись

Сведения по авторской справке подтверждаю: «____» _____2024 г.

Заведующая кафедрой ЭВМ: М. Л. Долженкова _____

подпись

Приложение Б

(обязательное)

Листинг кода

```
// токены

package token

type TokenType = string

type Token struct {
        Type    TokenType
        Literal string
}

type Pos struct {
        Line   int
        Offset int
}

const (
        ILLEGAL = "ILLEGAL"
        EOF     = "EOF"

        IDENT  = "IDENT"  // x, t, add
        INT    = "INT"    // 123
        STRING = "STRING" // "abcde"

        ASSIGN        = "="
        PLUS          = "+"
        MINUS         = "-"
        STAR          = "*"
        SLASH         = "/"
        EXCLAMINATION = "!"
        PERCENT       = "%"

        EQ  = "=="
        NEQ = "!="
        LEQ = "<="
        GEQ = ">="
```

```
        LT  = "<"
        GT  = ">"

        LAND = "&&"
        LOR  = "||"

        COMMA     = ","
        SEMICOLON = ";"
        COLON     = ":"

        LPAR     = "("
        RPAR     = ")"
        LBRACE   = "{"
        RBRACE   = "}"
        LBRACKET = "["
        RBRACKET = "]"

        // keywords
        IF     = "IF"
        ELSE   = "ELSE"
        TRUE   = "TRUE"
        FALSE  = "FALSE"
        FUNC   = "FUNCTION"
        RETURN = "RETURN"
)

var keywords = map[string]TokenType{
        "if":     IF,
        "else":   ELSE,
        "true":   TRUE,
        "false":  FALSE,
        "fn":     FUNC,
        "return": RETURN,
}

func LookupIdent(ident string) TokenType {
        if tok, ok := keywords[ident]; ok {
                return tok
        }
        return IDENT
}
```

```
// фрагменты листинга кода лексического анализатора!!!

package lexer

import "github.com/botscubes/bql/internal/token"

type Lexer struct {
        input   string
        ch      byte // current char
        pos     int  // current position (on current char)
        readPos int  // position after current char
        nlsemi  bool // if "true" '\n' translate to ';'
        loPos   token.Pos
}

func New(input string) *Lexer {
        l := &Lexer{
                input:  input,
                nlsemi: false,
                loPos: token.Pos{
                        Line:   1,
                        Offset: -1,
                },
        }

        l.readChar()
        return l
}

func (l *Lexer) NextToken() (token.Token, token.Pos) {
        l.skipWhitespace()

        nlsemi := false

        var tok token.Token
        switch l.ch {
        case '\n':
                tok = newToken(token.SEMICOLON, l.ch)
        case '=':
                if l.peekChar() == '=' {
                        l.readChar()
```

```
                        literal := "=="
                        tok = token.Token{Type: token.EQ, Literal:
literal}
                } else {
                        tok = newToken(token.ASSIGN, l.ch)
                }
        case '+':
                tok = newToken(token.PLUS, l.ch)
        case '-':
                tok = newToken(token.MINUS, l.ch)
        case '*':
                tok = newToken(token.STAR, l.ch)
        case '/':
                tok = newToken(token.SLASH, l.ch)
        case '!':
                if l.peekChar() == '=' {
                        l.readChar()
                        literal := "!="
                        tok = token.Token{Type: token.NEQ, Literal:
literal}
                } else {
                        tok = newToken(token.EXCLAMINATION, l.ch)
                }

        case '%':
                tok = newToken(token.PERCENT, l.ch)
        case '<':
                if l.peekChar() == '=' {
                        l.readChar()
                        literal := "<="
                        tok = token.Token{Type: token.LEQ, Literal:
literal}
                } else {
                        tok = newToken(token.LT, l.ch)
                }
        case '>':
                if l.peekChar() == '=' {
                        l.readChar()
                        literal := ">="
                        tok = token.Token{Type: token.GEQ, Literal:
literal}
```

```
                } else {
                        tok = newToken(token.GT, l.ch)
                }
        case ',':
                tok = newToken(token.COMMA, l.ch)
        case ';':
                tok = newToken(token.SEMICOLON, l.ch)
        case ':':
                tok = newToken(token.COLON, l.ch)
        case '(':
                tok = newToken(token.LPAR, l.ch)
        case ')':
                nlsemi = true
                tok = newToken(token.RPAR, l.ch)
        case '{':
                tok = newToken(token.LBRACE, l.ch)
        case '}':
                nlsemi = true
                tok = newToken(token.RBRACE, l.ch)
        case '[':
                tok = newToken(token.LBRACKET, l.ch)
        case ']':
                nlsemi = true
                tok = newToken(token.RBRACKET, l.ch)
        case '"':
                tok.Type = token.STRING
                tok.Literal = l.readString()
                nlsemi = true
        case '&':
                if l.peekChar() == '&' {
                        l.readChar()
                        literal := "&&"
                        tok = token.Token{Type: token.LAND, Literal:
literal}
                } else {
                        tok = newToken(token.ILLEGAL, l.ch)
                }
        case '|':
                if l.peekChar() == '|' {
                        l.readChar()
                        literal := "||"
```

```
                        tok = token.Token{Type: token.LOR, Literal:
   literal}
                } else {
                        tok = newToken(token.ILLEGAL, l.ch)
                }
        case 0:
                tok = token.Token{Type: token.EOF, Literal: ""}
        default:
                if isLetter(l.ch) {
                        tok.Literal = l.readIdent()
                        tok.Type = token.LookupIdent(tok.Literal)

                        if tok.Type == token.IDENT || tok.Type ==
   token.TRUE || tok.Type == token.FALSE {
                                l.nlsemi = true
                        }
                        return tok, l.loPos
                } else if isDigit(l.ch) {
                        tok.Type = token.INT
                        tok.Literal = l.readNumber()
                        l.nlsemi = true
                        return tok, l.loPos
                } else {
                        tok = newToken(token.ILLEGAL, l.ch)
                }
        }

        l.nlsemi = nlsemi

        l.readChar()
        return tok, l.loPos
}

func (l *Lexer) readChar() {
        if l.readPos >= len(l.input) {
                l.ch = 0 // EOF
        } else {
                l.ch = l.input[l.readPos]
        }

        l.pos = l.readPos
```

```
        l.loPos.Offset += 1
        l.readPos += 1
}

func (l *Lexer) peekChar() byte {
        if l.readPos >= len(l.input) {
                return 0
        } else {
                return l.input[l.readPos]
        }
}

func (l *Lexer) skipWhitespace() {
        for l.ch == ' ' || l.ch == '\n' && !l.nlsemi || l.ch == '\t'
   || l.ch == '\r' {
                l.readChar()

                if l.ch == '\n' {
                        l.onNewLine()
                }
        }
}

func isLetter(ch byte) bool {
        return 'a' <= ch && ch <= 'z' || 'A' <= ch && ch <= 'Z' || ch
    == '_'
}

func isDigit(ch byte) bool {
        return '0' <= ch && ch <= '9'
}

func (l *Lexer) readIdent() string {
        position := l.pos
        for isLetter(l.ch) || isDigit(l.ch) {
                l.readChar()
        }
        return l.input[position:l.pos]
}

func (l *Lexer) readNumber() string {
```

```
        position := l.pos
        for isDigit(l.ch) {
                l.readChar()
        }
        return l.input[position:l.pos]
}

func (l *Lexer) readString() string {
        position := l.pos + 1
        for {
                l.readChar()
                if l.ch == '"' || l.ch == 0 {
                        break
                }
        }
        return l.input[position:l.pos]
}

func newToken(tokenType token.TokenType, ch byte) token.Token {
        return token.Token{Type: tokenType, Literal: string(ch)}
}

func (l *Lexer) onNewLine() {
        l.loPos.Line += 1
        l.loPos.Offset = -1
}

// Фрагменты листинга кода синтаксического анализатора

package parser

import (
        "fmt"
        "strconv"

        "github.com/botscubes/bql/internal/ast"

        "github.com/botscubes/bql/internal/lexer"
        "github.com/botscubes/bql/internal/token"
)
```

```
const (
        _ int = iota
        LOWEST
        LOR         // ||
        LAND        // &&
        EQUALS      // ==
        LESSGREATER // > or < or <= or >=
        SUM         // +
        PRODUCT     // * % /
        PREFIX      // -x or !x
        CALL        // call(x) or ( expr )
        INDEX       // [
)

// TODO: create switch and move to token.go
var precedences = map[token.TokenType]int{
        token.LOR:      LOR,
        token.LAND:     LAND,
        token.EQ:       EQUALS,
        token.NEQ:      EQUALS,
        token.LT:       LESSGREATER,
        token.GT:       LESSGREATER,
        token.GEQ:      LESSGREATER,
        token.LEQ:      LESSGREATER,
        token.PLUS:     SUM,
        token.MINUS:    SUM,
        token.SLASH:    PRODUCT,
        token.STAR:     PRODUCT,
        token.PERCENT:  PRODUCT,
        token.LPAR:     CALL,
        token.LBRACKET: INDEX,
}

type (
        prefixParseFn func() ast.Expression
        infixParseFn  func(ast.Expression) ast.Expression
)

type Parser struct {
        l         *lexer.Lexer
        curToken  token.Token
```

```
        peekToken token.Token
        peekPos   token.Pos
        curPos    token.Pos

        prefixParsers map[token.TokenType]prefixParseFn
        infixParsers  map[token.TokenType]infixParseFn
        errors        []string
}

func New(l *lexer.Lexer) *Parser {
        p := &Parser{
                l: l,
        }

        // prefix parse functions
        p.prefixParsers = make(map[token.TokenType]prefixParseFn)
        p.prefixParsers[token.IDENT] = p.parseIdent
        p.prefixParsers[token.INT] = p.parseInteger
        p.prefixParsers[token.MINUS] = p.parsePrefixExpression
        p.prefixParsers[token.EXCLAMINATION] = p.
   parsePrefixExpression
        p.prefixParsers[token.TRUE] = p.parseBoolean
        p.prefixParsers[token.FALSE] = p.parseBoolean
        p.prefixParsers[token.LPAR] = p.parseGroupedExpression
        p.prefixParsers[token.IF] = p.parseIfExpression
        p.prefixParsers[token.STRING] = p.parseString
        p.prefixParsers[token.LBRACKET] = p.parseArray
        p.prefixParsers[token.FUNC] = p.parseFunction
        p.prefixParsers[token.LBRACE] = p.parseHashMapLiteral

        // infix parse functions
        p.infixParsers = make(map[token.TokenType]infixParseFn)
        p.infixParsers[token.PLUS] = p.parseInfixExpression
        p.infixParsers[token.MINUS] = p.parseInfixExpression
        p.infixParsers[token.STAR] = p.parseInfixExpression
        p.infixParsers[token.SLASH] = p.parseInfixExpression
        p.infixParsers[token.PERCENT] = p.parseInfixExpression
        p.infixParsers[token.EQ] = p.parseInfixExpression
        p.infixParsers[token.NEQ] = p.parseInfixExpression
        p.infixParsers[token.LEQ] = p.parseInfixExpression
        p.infixParsers[token.GEQ] = p.parseInfixExpression
```

```
        p.infixParsers[token.LT] = p.parseInfixExpression
        p.infixParsers[token.GT] = p.parseInfixExpression
        p.infixParsers[token.LOR] = p.parseInfixExpression
        p.infixParsers[token.LAND] = p.parseInfixExpression
        p.infixParsers[token.LPAR] = p.parseCallExpression
        p.infixParsers[token.LBRACKET] = p.parseIndexExpression

        // read curToken and peekToken
        p.nextToken()
        p.nextToken()

        return p
}

func (p *Parser) Errors() []string {
        return p.errors
}

func (p *Parser) newError(e string) {
        mes := fmt.Sprintf("pos: %d:%d: %s", p.curPos.Line, p.curPos.
   Offset, e)
        p.errors = append(p.errors, mes)
}

func (p *Parser) nextToken() {
        p.curToken = p.peekToken
        p.curPos = p.peekPos
        p.peekToken, p.peekPos = p.l.NextToken()
}

func (p *Parser) curTokenIs(t token.TokenType) bool {
        return p.curToken.Type == t
}

func (p *Parser) peekTokenIs(t token.TokenType) bool {
        return p.peekToken.Type == t
}

func (p *Parser) peekPrecedence() int {
        if p, ok := precedences[p.peekToken.Type]; ok {
                return p
```

```
        }

        return LOWEST
}

func (p *Parser) expectPeek(t token.TokenType) bool {
        if p.peekTokenIs(t) {
                p.nextToken()
                return true
        } else {
                p.newError(fmt.Sprintf("expected next token: %s, got
   %s", t, p.peekToken.Type))
                return false
        }
}

func (p *Parser) skipPeek(t token.TokenType) bool {
        if p.peekTokenIs(t) {
                p.nextToken()
                return true
        } else {
                return false
        }
}

func (p *Parser) curPrecedence() int {
        if p, ok := precedences[p.curToken.Type]; ok {
                return p
        }

        return LOWEST
}

func (p *Parser) expectSemi() bool {
        if !p.curTokenIs(token.RBRACE) && !p.curTokenIs(token.EOF) {
                if !p.curTokenIs(token.SEMICOLON) {
                        p.newError(fmt.Sprintf("expected ; at end of
   statement, got %s", p.curToken.Literal))
                        return false
                }
                p.nextToken()
```

```
        }
        return true
}

func (p *Parser) ParseProgram() *ast.Program {
        program := &ast.Program{}
        program.Statements = []ast.Statement{}

        for !p.curTokenIs(token.EOF) {
                stmt := p.parseStatement()
                if stmt != nil {
                        program.Statements = append(program.
   Statements, stmt)
                }
                p.nextToken()

                if ok := p.expectSemi(); !ok {
                        break
                }

        }

        return program
}

func (p *Parser) parseStatement() ast.Statement {
        switch p.curToken.Type {
        case token.IDENT:
                if p.peekTokenIs(token.ASSIGN) {
                        return p.parseAssignStatement()
                }

                return p.parseExpressionStatement()
        case token.RETURN:
                return p.parseReturnStatement()
        default:
                return p.parseExpressionStatement()
        }
}

func (p *Parser) parseAssignStatement() *ast.AssignStatement {
```

```
        stmt := &ast.AssignStatement{
                Name: &ast.Ident{Token: p.curToken, Value: p.curToken
   .Literal},
        }

        // skip ident and =
        p.nextToken()
        p.nextToken()

        stmt.Value = p.parseExpression(LOWEST)

        return stmt
}

func (p *Parser) parseExpressionStatement() *ast.ExpressionStatement
   {
        stmt := &ast.ExpressionStatement{Token: p.curToken}

        stmt.Expression = p.parseExpression(LOWEST)

        return stmt
}

func (p *Parser) parseBlockStatement() *ast.BlockStatement {
        block := &ast.BlockStatement{Token: p.curToken}
        block.Statements = []ast.Statement{}

        p.nextToken()

        for !p.curTokenIs(token.RBRACE) && !p.curTokenIs(token.EOF) {
                stmt := p.parseStatement()
                if stmt != nil {
                        block.Statements = append(block.Statements,
   stmt)
                }
                p.nextToken()

                if ok := p.expectSemi(); !ok {
                        break
                }
        }
```

```
        return block
}

func (p *Parser) parseReturnStatement() *ast.ReturnStatement {
        stmt := &ast.ReturnStatement{Token: p.curToken}

        p.nextToken()

        stmt.Value = p.parseExpression(LOWEST)

        if p.peekTokenIs(token.SEMICOLON) {
                p.nextToken()
        }

        return stmt
}

func (p *Parser) parseExpression(precedence int) ast.Expression {
        prefix := p.prefixParsers[p.curToken.Type]
        if prefix == nil {
                p.newError(fmt.Sprintf("prefix parse function for %s
   not found", p.curToken.Type))
                return nil
        }
        leftExp := prefix()

        for !p.peekTokenIs(token.SEMICOLON) && precedence < p.
   peekPrecedence() {
                infix := p.infixParsers[p.peekToken.Type]
                if infix == nil {
                        return leftExp
                }

                p.nextToken()

                leftExp = infix(leftExp)
        }

        return leftExp
}
```

```
func (p *Parser) parseFunction() ast.Expression {
        node := &ast.FunctionLiteral{Token: p.curToken}

        if !p.expectPeek(token.LPAR) {
                return nil
        }

        node.Parameters = p.parseFunctionParameters()

        if !p.expectPeek(token.LBRACE) {
                return nil
        }

        node.Body = p.parseBlockStatement()

        return node
}

// Листинг кода объектной системы

package object

import (
        "bytes"
        "fmt"
        "hash/fnv"
        "strings"

        "github.com/botscubes/bql/internal/ast"
)

type ObjectType string

type BuiltinFunction func(args ...Object) Object

type Object interface {
        Type() ObjectType
        ToString() string
}
```

```
const (
        ERROR_OBJ = "ERROR"
        NULL_OBJ  = "NULL"

        RETURN_VALUE_OBJ = "RETURN_VALUE"

        INTEGER_OBJ  = "INTEGER"
        BOOLEAN_OBJ  = "BOOLEAN"
        STRING_OBJ   = "STRING"
        ARRAY_OBJ    = "ARRAY"
        HASH_MAP_OBJ = "HASH_MAP"

        FUNCTION_OBJ = "FUNCTION"
        BUILTIN_OBJ  = "BUILTIN"
)

type HashKey struct {
        Type  ObjectType
        Value uint64
}

type Hashable interface {
        HashKey() HashKey
}

type Null struct{}

func (n *Null) Type() ObjectType { return NULL_OBJ }
func (n *Null) ToString() string { return "Null" }

type Error struct {
        Message string
}

func (e *Error) Type() ObjectType { return ERROR_OBJ }
func (e *Error) ToString() string { return "error: " + e.Message }

type Return struct {
        Value Object
}
```

```
func (r *Return) Type() ObjectType { return RETURN_VALUE_OBJ }
func (r *Return) ToString() string { return r.Value.ToString() }

type Integer struct {
        Value int64
}

func (i *Integer) Type() ObjectType { return INTEGER_OBJ }
func (i *Integer) ToString() string { return fmt.Sprintf("%d", i.
   Value) }
func (i *Integer) HashKey() HashKey { return HashKey{Type: i.Type(),
   Value: uint64(i.Value)} }

type Boolean struct {
        Value bool
}

func (b *Boolean) Type() ObjectType { return BOOLEAN_OBJ }
func (b *Boolean) ToString() string { return fmt.Sprintf("%t", b.
   Value) }
func (b *Boolean) HashKey() HashKey {
        if b.Value {
                return HashKey{Type: b.Type(), Value: 1}
        }
        return HashKey{Type: b.Type(), Value: 0}
}

type Function struct {
        Parameters []*ast.Ident
        Body       *ast.BlockStatement
        Env        *Env
}

func (f *Function) Type() ObjectType { return FUNCTION_OBJ }
func (f *Function) ToString() string {
        var out bytes.Buffer

        params := []string{}
        for _, p := range f.Parameters {
                params = append(params, p.ToString())
        }
```

```
        out.WriteString("fn")
        out.WriteString("(")
        out.WriteString(strings.Join(params, ", "))
        out.WriteString(") ")
        out.WriteString("{ ")
        out.WriteString(f.Body.ToString())
        out.WriteString("} ")

        return out.String()
}

type String struct {
        Value string
}

func (s *String) Type() ObjectType { return STRING_OBJ }
func (s *String) ToString() string { return s.Value }
func (s *String) HashKey() HashKey {
        h := fnv.New64a()
        h.Write([]byte(s.Value))

        return HashKey{Type: s.Type(), Value: h.Sum64()}
}

type Array struct {
        Elements []Object
}

func (a *Array) Type() ObjectType { return ARRAY_OBJ }
func (a *Array) ToString() string {
        var out bytes.Buffer

        elements := []string{}
        for _, el := range a.Elements {
                elements = append(elements, el.ToString())
        }

        out.WriteString("[")
        out.WriteString(strings.Join(elements, ", "))
        out.WriteString("]")
```

```
        return out.String()
}

type HashPair struct {
        Key   Object
        Value Object
}

type HashMap struct {
        Pairs map[HashKey]HashPair
}

func (h *HashMap) Type() ObjectType { return HASH_MAP_OBJ }
func (h *HashMap) ToString() string {
        var out bytes.Buffer

        pairs := []string{}
        for _, el := range h.Pairs {
                pairs = append(pairs, fmt.Sprintf("%s: %s", el.Key.
   ToString(), el.Value.ToString()))
        }

        out.WriteString("[")
        out.WriteString(strings.Join(pairs, ", "))
        out.WriteString("]")

        return out.String()
}

type Builtin struct {
        Fn BuiltinFunction
}

func (b *Builtin) Type() ObjectType { return BUILTIN_OBJ }
func (b *Builtin) ToString() string { return "builtin function" }

// Фрагменты листинга кода семантического анализатора

func newError(formating string, parameters ...any) *object.Error {
        return &object.Error{Message: fmt.Sprintf(formating,
```

```
   parameters...)}
}

func isError(obj object.Object) bool {
        if obj != nil {
                return obj.Type() == object.ERROR_OBJ
        }
        return false
}

func evalPrefixExpression(op string, right object.Object) object.
   Object {
        switch op {
        case "!":
                return evalExclOpExpr(right)
        case "-":
                return evalMinusPrefixOpExpr(right)
        default:
                return newError("unknown operator: %s", op)
        }
}


func evalInfixExpression(op string, left object.Object, right object.
   Object) object.Object {
        switch {
        case left.Type() == object.INTEGER_OBJ && right.Type() ==
   object.INTEGER_OBJ:
                return evalIntInfixExpr(op, left, right)
        case left.Type() == object.STRING_OBJ && right.Type() ==
   object.STRING_OBJ:
                return evalStringInfixExpr(op, left, right)
        case op == "==":
                return boolToBooleanObj(left == right)
        case op == "!=":
                return boolToBooleanObj(left != right)
        case op == "||":
                return boolToBooleanObj(left.(*object.Boolean).Value
   || right.(*object.Boolean).Value)
        case op == "&&":
                return boolToBooleanObj(left.(*object.Boolean).Value
```

```
   && right.(*object.Boolean).Value)
        case left.Type() != right.Type():
                return newError("type mismatch: %s %s %s", left.Type
   (), op, right.Type())
        default:
                return newError("unknown operator: %s %s %s", left.
   Type(), op, right.Type())
        }
}

func evalIfExpression(node *ast.IfExpression, env *object.Env) object
   .Object {
        condition := Eval(node.Condition, env)
        if isError(condition) {
                return condition
        }

        if condition != TRUE && condition != FALSE {
                return newError("non boolean condition in if
   statement")
        }

        if condition == TRUE {
                return Eval(node.Consequence, env)
        } else if node.Alternative != nil {
                return Eval(node.Alternative, env)
        } else {
                return NULL
        }
}

func evalIndexExpression(left, index object.Object) object.Object {
        switch {
        case left.Type() == object.ARRAY_OBJ && index.Type() ==
   object.INTEGER_OBJ:
                return evalArrayIndexExp(left, index)
        case left.Type() == object.HASH_MAP_OBJ:
                return evalHashMapIndexExp(left, index)
        default:
                return newError("index operator not supported: %s",
   left.Type())
```

```
        }
}

// Фрагменты листинга кода исполнителя

package evaluator

import (
        "fmt"

        "github.com/botscubes/bql/internal/ast"
        "github.com/botscubes/bql/internal/object"
)

var (
        TRUE  = &object.Boolean{Value: true}
        FALSE = &object.Boolean{Value: false}
        NULL  = &object.Null{}
)

func boolToBooleanObj(b bool) *object.Boolean {
        if b {
                return TRUE
        }

        return FALSE
}

func Eval(n ast.Node, env *object.Env) object.Object {
        switch node := n.(type) {
        case *ast.Program:
                return evalProgram(node, env)

        case *ast.BlockStatement:
                return evalBlockStatement(node, env)

        case *ast.ExpressionStatement:
                return Eval(node.Expression, env)

        case *ast.ReturnStatement:
                val := Eval(node.Value, env)
```

```
            if isError(val) {
                    return val
            }

            return &object.Return{Value: val}

    case *ast.AssignStatement:
            val := Eval(node.Value, env)
            if isError(val) {
                    return val
            }

            env.Set(node.Name.Value, val)

    case *ast.IntegerLiteral:
            return &object.Integer{Value: node.Value}

    case *ast.Boolean:
            if node.Value {
                    return TRUE
            }

            return FALSE

    case *ast.StringLiteral:
            return &object.String{Value: node.Value}

    case *ast.PrefixExpression:
            right := Eval(node.Right, env)
            if isError(right) {
                    return right
            }

            return evalPrefixExpression(node.Operator, right)

    case *ast.InfixExpression:
            left := Eval(node.Left, env)
            if isError(left) {
                    return left
            }
```

```
                right := Eval(node.Right, env)
                if isError(right) {
                        return right
                }

                return evalInfixExpression(node.Operator, left, right
)

        case *ast.IfExpression:
                return evalIfExpression(node, env)

        case *ast.Ident:
                return evalIdent(node, env)

        case *ast.FunctionLiteral:
                return &object.Function{Parameters: node.Parameters,
Body: node.Body, Env: env}

        case *ast.CallExpression:
                function := Eval(node.FnName, env)
                if isError(function) {
                        return function
                }

                args := evalExpressions(node.Arguments, env)
                if len(args) == 1 && isError(args[0]) {
                        return args[0]
                }

                return callFunction(function, args)

        case *ast.ArrayLiteral:
                elements := evalExpressions(node.Elements, env)
                if len(elements) == 1 && isError(elements[0]) {
                        return elements[0]
                }

                return &object.Array{Elements: elements}

        case *ast.IndexExpression:
                left := Eval(node.Left, env)
```

```
                if isError(left) {
                        return left
                }

                index := Eval(node.Index, env)
                if isError(index) {
                        return index
                }

                return evalIndexExpression(left, index)
        case *ast.HashMapLiteral:
                return evalHashMap(node, env)
        }

        return nil
}

func evalProgram(program *ast.Program, env *object.Env) object.Object
    {
        var result object.Object

        for _, stmt := range program.Statements {
                result = Eval(stmt, env)

                switch r := result.(type) {
                case *object.Return:
                        return r.Value
                case *object.Error:
                        return r
                }
        }

        return result
}

func evalIntInfixExpr(op string, left object.Object, right object.
   Object) object.Object {
        lVal := left.(*object.Integer).Value
        rVal := right.(*object.Integer).Value
        switch op {
        case "+":
```

```
                return &object.Integer{Value: lVal + rVal}
        case "-":
                return &object.Integer{Value: lVal - rVal}
        case "*":
                return &object.Integer{Value: lVal * rVal}
        case "/":
                return &object.Integer{Value: lVal / rVal}
        case "%":
                return &object.Integer{Value: lVal % rVal}
        case "==":
                return boolToBooleanObj(lVal == rVal)
        case "!=":
                return boolToBooleanObj(lVal != rVal)
        case "<":
                return boolToBooleanObj(lVal < rVal)
        case ">":
                return boolToBooleanObj(lVal > rVal)
        case "<=":
                return boolToBooleanObj(lVal <= rVal)
        case ">=":
                return boolToBooleanObj(lVal >= rVal)
        default:
                return newError("unknown operator: %s %s %s", left.
   Type(), op, right.Type())
        }
}

func evalIdent(node *ast.Ident, env *object.Env) object.Object {
        if val, ok := env.Get(node.Value); ok {
                return val
        }

        if builtin, ok := builtins[node.Value]; ok {
                return builtin
        }

        return newError("identifier not found: " + node.Value)
}

func callFunction(function object.Object, args []object.Object)
   object.Object {
```

```
        switch fn := function.(type) {
        case *object.Function:
                extEnv := extendFuncEnv(fn, args)
                ev := Eval(fn.Body, extEnv)
                return unwrapReturn(ev)
        case *object.Builtin:
                return fn.Fn(args...)
        default:
                return newError("call not a function: %s", fn.Type())
        }
}

func extendFuncEnv(fn *object.Function, args []object.Object) *object
   .Env {
        env := object.NewEnclosedEnv(fn.Env)

        for id, param := range fn.Parameters {
                env.Set(param.Value, args[id])
        }

        return env
}

func unwrapReturn(obj object.Object) object.Object {
        if val, ok := obj.(*object.Return); ok {
                return val.Value
        }

        return obj
}

func evalHashMap(node *ast.HashMapLiteral, env *object.Env) object.
   Object {
        pairs := make(map[object.HashKey]object.HashPair)

        for knode, vnode := range node.Pairs {
                key := Eval(knode, env)
                if isError(key) {
                        return key
                }
```

```
                hashKey, ok := key.(object.Hashable)
                if !ok {
                        return newError("unusable as hash key: %s",
   key.Type())
                }

                value := Eval(vnode, env)
                if isError(value) {
                        return value
                }

                pairs[hashKey.HashKey()] = object.HashPair{Key: key,
   Value: value}
        }

        return &object.HashMap{Pairs: pairs}
}

func evalHashMapIndexExp(hashMap, index object.Object) object.Object
   {
        hashObject := hashMap.(*object.HashMap)

        key, ok := index.(object.Hashable)
        if !ok {
                return newError("unusable as hash key: %s", index.
   Type())
        }

        pair, ok := hashObject.Pairs[key.HashKey()]
        if !ok {
                return NULL
        }

        return pair.Value
}

// Фрагменты листинга кода тестов
func TestEvalIntegerExpression(t *testing.T) {
        tests := []struct {
                input    string
                expected int64
```

```
        }{
                {"4", 4},
                {"12", 12},
                {"-5", -5},
                {"-15", -15},
                {"2 + 2", 4},
                {"4 - 2", 2},
                {"2 - 2", 0},
                {"1 + 2 + 3 + 4 + 5 - 1 - 2 - 3", 9},
                {"2 * 3 * 4 * 5 * 6 * 7 * 8 * 9", 362880},
                {"10 + 10 * 2", 30},
                {"(10 + 10) * 2", 40},
                {"100 / 2 * 2 + 5", 105},
                {"100 / (2 * 2) - 200", -175},
                {"5 % 2", 1},
                {"4 % 2", 0},
        }

        for _, test := range tests {
                ev := getEvaluated(test.input)
                testInteger(t, ev, test.expected)
        }
}

func TestEvalBooleanExpresion(t *testing.T) {
        tests := []struct {
                input    string
                expected bool
        }{
                {"true", true},
                {"false", false},
                {"1 == 1", true},
                {"1 != 1", false},
                {"1 < 2", true},
                {"1 > 2", false},
                {"1 <= 2", true},
                {"1 <= 1", true},
                {"1 >= 2", false},
                {"1 >= 1", true},
                {"true == true", true},
                {"false == false", true},
```

```
                {"true == false", false},
                {"true != false", true},
                {"(true == false) == false", true},
                {"(1 == 1) == true", true},
                {"(2 > 1) == true", true},
                {"(2 < 1) == false", true},
                {"(1 <= 1) == false", false},
                {"true || false", true},
                {"true && false", false},
                {"true && true", true},
                {"false && false", false},
                {"false || false", false},
        }

        for _, test := range tests {
                ev := getEvaluated(test.input)
                testBoolean(t, ev, test.expected, test.input)
        }
}

func TestExclaminationOperator(t *testing.T) {
        tests := []struct {
                input    string
                expected bool
        }{
                {"!true", false},
                {"!false", true},
                {"!!false", false},
                {"!!true", true},
        }

        for _, test := range tests {
                ev := getEvaluated(test.input)
                testBoolean(t, ev, test.expected, test.input)
        }
}

func TestIfElseExpression(t *testing.T) {
        tests := []struct {
                input    string
                expected any
```

```
        }{
                {"if (true) { 50 }", 50},
                {"if (false) { 50 }", nil},
                {"if (!false) { 50 }", 50},
                {"if (1 == 1) { 50 }", 50},
                {"if (1 > 2) { 50 }", nil},
                {"if (1 < 2) { 50 }", 50},
                {"if (1 < 2) { 50 } else { 100 }", 50},
                {"if (1 > 2) { 50 } else { 100 }", 100},
                {"if (true || false) { 50 } else { 100 }", 50},
                {"if (true && false) { 50 } else { 100 }", 100},
        }

        for _, test := range tests {
                ev := getEvaluated(test.input)
                intVal, ok := test.expected.(int)
                if ok {
                        testInteger(t, ev, int64(intVal))
                } else {
                        testNull(t, ev)
                }
        }
}

func TestEvalAssignExpression(t *testing.T) {
        tests := []struct {
                input    string
                expected int64
        }{
                {"x = 10; x", 10},
                {"x = 10; x = 15; x", 15},
                {"x = 10 * 2; x", 20},
                {"x = 10 * 2; y = x * 3; y", 60},
                {"x = 10; y = x * 2; z = x + y - 20; z", 10},
        }

        for _, test := range tests {
                ev := getEvaluated(test.input)
                testInteger(t, ev, test.expected)
        }
}
```

```
func TestEvalStringExpression(t *testing.T) {
        tests := []struct {
                input    string
                expected string
        }{
                {`"Hello Earth"`, "Hello Earth"},
                {`"Hello" + " " + "Earth"`, "Hello Earth"},
        }

        for _, test := range tests {
                ev := getEvaluated(test.input)
                res, ok := ev.(*object.String)
                if !ok {
                        t.Errorf("obj not String got:%+v", ev)
                        return
                }

                if res.Value != test.expected {
                        t.Errorf("obj wrong value. got: %s expected:
   %s", res.Value, test.expected)
                }
        }
}

func TestArray(t *testing.T) {
        input := "[1, 2, -33, 5+5, 1 + 2 + 3 + 4 * 5]"

        ev := getEvaluated(input)
        res, ok := ev.(*object.Array)
        if !ok {
                t.Errorf("obj not Array got:%+v", ev)
                return
        }

        if len(res.Elements) != 5 {
                t.Errorf("wrong array length got:%d expected 5", len(
   res.Elements))
                return
        }
```

```
        testInteger(t, res.Elements[0], 1)
        testInteger(t, res.Elements[1], 2)
        testInteger(t, res.Elements[2], -33)
        testInteger(t, res.Elements[3], 10)
        testInteger(t, res.Elements[4], 26)
}

func TestArrayIndexExpression(t *testing.T) {
        tests := []struct {
                input    string
                expected any
        }{
                {"[1, 2, 5][0]", 1},
                {"[1, 2, 5][2]", 5},
                {"[1, 2, 5][3]", nil},
                {"[1, 2, 5][-1]", nil},
                {"[1, 2, 5][1+1]", 5},
                {"x = 1; [1, 2, 5][x]", 2},
                {"a = [1, 2, 5]; a[0] + a[1] * a[2]", 11},
        }

        for _, test := range tests {
                ev := getEvaluated(test.input)
                intVal, ok := test.expected.(int)
                if ok {
                        testInteger(t, ev, int64(intVal))
                } else {
                        testNull(t, ev)
                }
        }
}

func TestHashMap(t *testing.T) {
        input := `x = "v";
        {
                "a": 1,
                "bb": 10*10,
                x: 4,
                "qq"+"ww": 123,
                true: 1,
                false: 0
```

```
        }`

        ev := getEvaluated(input)
        res, ok := ev.(*object.HashMap)
        if !ok {
                t.Fatalf("non HashMap returned: %T - %+v", ev, ev)
        }

        expected := map[object.HashKey]int64{
                (&object.String{Value: "a"}).HashKey():    1,
                (&object.String{Value: "bb"}).HashKey():   100,
                (&object.String{Value: "v"}).HashKey():    4,
                (&object.String{Value: "qqww"}).HashKey(): 123,
                TRUE.HashKey():  1,
                FALSE.HashKey(): 0,
        }

        if len(res.Pairs) != len(expected) {
                t.Fatalf("wrong len pairs. got: %d expected:%d", len(
   res.Pairs), len(expected))
        }

        for ek, ev := range expected {
                p, ok := res.Pairs[ek]
                if !ok {
                        t.Errorf("not found pair for Key")
                }

                testInteger(t, p.Value, ev)
        }
}

func TestBuiltinFunctions(t *testing.T) {
        tests := []struct {
                input    string
                expected any
        }{
                {`len("abc")`, 3},
                {`len("abc" + "efg")`, 6},
                {`len("")`, 0},
                {`len(1)`, "type of argument not supported: INTEGER"
```

```
    },
                {`len("a", "b")`, "wrong number of arguments: 2 want:
     1"},
                {`x = "abc"; len(x)`, 3},
                {`len([])`, 0},
                {`len([1, 2])`, 2},
                {`x = [1, 2, 3]; len(x)`, 3},
                {`push([], 4)`, []int{4}},
                {`push([1, 2, 3], 4)`, []int{1, 2, 3, 4}},
                {`push("a", 4)`, "first argument must be ARRAY, got:
    STRING"},
                {`first([])`, nil},
                {`first([1])`, 1},
                {`first([3, 2, 1])`, 3},
                {`first("a")`, "argument must be ARRAY, got: STRING"
    },
                {`last([])`, nil},
                {`last([1])`, 1},
                {`last([3, 2, 1])`, 1},
                {`last("a")`, "argument must be ARRAY, got: STRING"},
        }

        for _, test := range tests {
                ev := getEvaluated(test.input)
                switch ex := test.expected.(type) {
                case int:
                        testInteger(t, ev, int64(ex))
                case nil:
                        testNull(t, ev)
                case string:
                        res, ok := ev.(*object.Error)
                        if !ok {
                                t.Errorf("object is not Error: %T -
    %+v", ev, ev)
                                continue
                        }

                        if res.Message != ex {
                                t.Errorf("wrong error message: %s
    expected: %s", res.Message, ex)
                        }
```

```
                case []int:
                        res, ok := ev.(*object.Array)
                        if !ok {
                                t.Errorf("object is not Array: %T -
   %+v", ev, ev)
                                continue
                        }

                        if len(res.Elements) != len(ex) {
                                t.Errorf("wrong number of elements: %
   d expected: %d", len(res.Elements), len(ex))
                                continue
                        }

                        for i, el := range ex {
                                testInteger(t, res.Elements[i], int64
   (el))
                        }
                }
        }
}
```

Приложение В

(справочное)

Список сокращений и обозначений

РБНФ – расширенная Бэкуса-Наура форма

API – application programming interface (программный интерфейс приложения)

AST – abstract syntax tree (абстрактное синтаксическое дерево)

DSL – domain-specific language (предметно-ориентированный язык)

MIT – MIT License/X11 License – лицензия свободного программного обеспечения, разработанная Массачусетским технологическим институтом

Приложение Г

(справочное)

Библиографический список

1. Low-code и No-code: как программировать без кода [Электронный ресурс]. – Режим доступа: https://blog.sf.education/low-code-i-no-code/

2. Предметно-ориентированный язык — Википедия [Электронный ресурс]. – Режим доступа: https://ru.wikipedia.org/wiki/Предметно-ориентированный_язык

3. Классификация предметно-ориентированных языков и языковых инструментариев [Электронный ресурс] – Режим доступа: https://www.hse.ru/data/2013/01/21/1305680244/Сухов-Классификация.pdf.

4. Вл. Пономарев. Конспективное изложение теории языков программирования и методов трансляции. Учебно-методическое пособие. В 4-х книгах. Книга 1. Формальные языки и грамматики [Текст]. – Озерск: ОТИ НИЯУ МИФИ, 2019. – 42 с.: ил.

5. Расширенная форма Бэкуса — Наура — Википедия [Электронный ресурс]. – Режим доступа: https://ru.wikipedia.org/wiki/Расширенная_форма_Бэкуса_—_Наура.

6. Молчанов А. Ю. Системное программное обеспечение. Учебник для вузов. 3-е изд [Текст]. – СПб.: Питер, 2010. – 400 с.: ил.

7. Ахо А. В. Компиляторы: принципы, технологии и инструментарий [Текст] / А. В. Ахо, М. С. Лам, Р. Сети, Д. Д. Ульман. – 2-е изд. – М: Вильямс, 2018. – 1184 с.: ил.

8. Ганичева О.Г. Теория языков программирования и методы трансляции. Учебное пособие [Текст]. – Череповец: ЧГУ, 2011. – 185 с.: ил.

9. Go — Википедия [Электронный ресурс]. – Режим доступа: https://ru.wikipedia.org/wiki/Go.

10. Различные виды тестирования программного обеспечения | AppMaster [Электронный ресурс]. – Режим доступа: https://appmaster.io/ru/blog/vidy-testirovaniia-programmnogo-obespecheniia.

11. Тестирование в Go | AppMaster [Электронный ресурс]. – Режим доступа: https://appmaster.io/ru/blog/testirovanie-v-go.