МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Вятский государственный университет»
Факультет автоматики и вычислительной техники
Кафедра электронных вычислительных машин

Допущено к защите
Руководитель проекта
_____ (Долженкова М. Л.)
«___» _____2024г.

Разработка модуля интерпретации предметно-ориентированного языка

Пояснительная записка курсового проекта по дисциплине
«Комплекс знаний бакалавра в области программного и аппаратного обеспечения вычислительной техники»
ТПЖА.090301.331 ПЗ

Разработал студент группы ИВТ-41 _____/Крючков И. С./

Руководитель _____/Долженкова М. Л./

Консультант _____/Кошкин О. В./

Проект защищен с оценкой «_____» _____
(оценка) (дата)

Члены комиссии _____/_____/
(подпись) (Ф.И.О.)

_____/_____/
(подпись) (Ф.И.О.)

_____/_____/
(подпись) (Ф.И.О.)

Киров 2024

Реферат

Крючков И. С. Разработка модуля интерпретации предметно-ориентированного языка. ТПЖА.090301.331 ПЗ: Курс. проект / ВятГУ, каф. ЭВМ; рук. Долженкова М. Л. - Киров, 2024. – ПЗ 63 с, 19 рис., 13 табл., 4 источника, 8 прил.

ПРЕДМЕТНО-ОРИЕНТИРОВАННЫЙ ЯЗЫК, ГРАММАТИКА, СЕМАНТИЧЕСКИЙ АНАЛИЗ, МОДУЛЬ ИНТЕРПРЕТАЦИИ, ИСПОЛНИТЕЛЬ, КОНСТРУКТОР, GO.

Объект курсового проекта – предметно-ориентированный язык.

Предмет – модуль интерпретации.

Цель курсового проекта – создать язык для расширения функциональных возможностей визуального конструктора.

Результат работы – разработанный модуль интерпретации предметно-ориентированного языка, расширяющего функциональные возможности визуального конструктора.

Содержание

Введение

В современном мире все больше обретают популярность сервисы визуального конструирования приложений. Конструктор позволяет широкому кругу пользователей создавать программные продукты с помощью визуального редактора. Однако функциональные возможности визуального редактора ограничены и их зачастую недостаточно для построения сложных, специфичных приложений. Расширение возможностей платформы возможно за счёт использования предметно-ориентированного языка программирования для конструктора. С помощью написания инструкций на данном языке, пользователи конструктора могут гибко задавать логику работы разрабатываемого приложения.

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.

1.1 Актуальность темы

Конструктором называется NoCode/LowCode инструмент, предназначенный для быстрого создания ботов без обязательного знания языков программирования общего назначения. Иными словами, весь процесс создания – это взаимодействие с визуальными компонентами платформы для построения логики работы бота.

Однако использование только визуальных инструментов конструктора накладывает некоторые функциональные ограничения. Чтобы создание бота было более гибким, в конструктор можно интегрировать предметно-ориентированный язык программирования, направленный на расширение функциональных возможностей визуального редактора.

Первое предназначение – упрощение работы. Ведь не все обладают знаниями и навыками программирования, достаточными для создания даже простых программных продуктов.

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.

Несмотря на все преимущества, у визуальных конструкторов есть значительные ограничения, которые не позволяют гибко настраивать логику

работы разрабатываемого приложения. Визуальные конструкторы обычно предоставляют набор готовых блоков, что может быть недостаточным для решения сложных задач. Специальный язык программирования для конструктора представляет собой одно из решений для преодоления данных ограничений. В отличие от графического интерфейса – язык позволяет пользователям сервиса гибко описывать логику работы приложения.

Предметно-ориентированный язык программирования (domain-specific language, DSL) – язык, специализированный для конкретной предметной области применения. Противоположностью DSL являются языки общего назначения, такие как C++, python, Go и т.д. Пример предметно-ориентированного языка – язык запросов SQL, который применяется при работе с базами данных.

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, 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. Также дополнительной трудностью является необходимость обучения разработчиков использованию языка.

В ранее выполненных курсовых проектах была разработана грамматика предметно-ориентированного языка, алгоритмы функционирования лексического и синтаксического анализаторов, осуществлена их программная реализация и тестирование. В результате был разработан лексический и синтаксический анализатор предметно-ориентированного языка.

1.2 Постановка задачи

1.2.1 Основание для разработки

Предметно-ориентированный язык визуального конструктора разрабатывается на основе задания на курсовое проектирование, полученного от компании ООО «Синаптик», г. Киров.

1.2.2 Цель и задачи разработки

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

Задачи:

– разработать программу, включающую в себя семантический анализатор для проверки семантической корректности конструкций языка и этап вычисления выражений, сформированных на этапе синтаксического анализа;

– выполнить тестирование разработанной программы с целью подтверждения корректности её работы.

1.2.3 Определение ключевых конструкция языка

Предметно-ориентированный язык конструктора должен включать следующие ключевые конструкции: переменные, ветвления, функции.

Поддерживаемые типы данных: строки, числа, булевы значения.

Кроме этого, язык должен поддерживать составные типы данных – массив, хэш-карта.

Набор основных 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.

Вывод

В данном разделе был проведен анализ предметной области и основываясь на ранее выполненных курсовых проектах по разработке грамматики, лексического и синтаксического анализаторов предметно-ориентированного языка, определены основные требования к разрабатываемому программному продукту.

Визуальные конструкторы значительно упрощают процесс создания и запуска приложений, однако, они имеет функциональные ограничения, которые препятствуют разработке продукта со сложной логикой работы. Предметно-ориентированный язык позволяет расширить возможности визуального конструктора и создавать уникальные приложения, в соответствии с заданными потребностями.

Таким образом, проблема является актуальной, и разработка модуля интерпретации предметно-ориентированного языка является актуальной.

2 Особенности этапов лексического и синтаксического анализа

В предыдущем курсовом проекте была разработана формальная грамматика предметно-ориентированного языка, а также лексический и синтаксический анализаторы.

Существует несколько способов описания синтаксиса языков программирования, например такие как формы Бэкуса-Наура, диаграммы Вирта и т.д.

В данном проекте для описания грамматики языка используется расширенная форма Бэкуса-Наура (РБНФ).

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

Формальная грамматика задаётся четвёркой вида:

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

где 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.;

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;

P – множество правил вывода грамматики;

S – начальный символ грамматики, $S \in V_N$.

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

Описание формальной грамматики приведено в приложении Е.

Помимо грамматики были разработаны лексический и синтаксический анализаторы языка.

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.

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.

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. Синтаксическое дерево подается на вход модулю интерпретации предметно-ориентированного языка.

Вывод

В данном разделе были описаны этапы, выполненные в предыдущем курсовом проекте, а именно, разработана формальная грамматика предметно-ориентированного языка, лексический и синтаксический анализаторы. Разработанные анализаторы являются необходимой составляющей для выполнения поставленных задач этого курсового проекта.

Синтаксический анализатор генерирует абстрактное синтаксическое дерево, которое необходимо, для работы семантического анализатора и исполнителя. Таким образом результаты выполнения предыдущего курсового проекта являются неотъемлемой частью дальнейшей разработки.

3 Разработка интерпретатора

На данном этапе необходимо в соответствие с поставленными требованиями разработать ключевые компоненты транслятора языка – семантический анализатор и механизм вычисления выражений программы – исполнитель (evaluator).

Некоторые фазы транслятора языка могут быть сгруппированы – то есть выполняется за один проход. Так этап семантического анализа выполняется непосредственно перед исполнением выражений программы.

3.1 Разработка семантического анализатора

В данном разделе необходимо выполнить разработку алгоритмов функционирования и программную реализацию семантического анализатора.

3.1.1 Разработка алгоритмов функционирования

Семантический анализ – это этап работы транслятора, вот 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.

Обобщенная структура интерпретатора показана на рисунке 1.

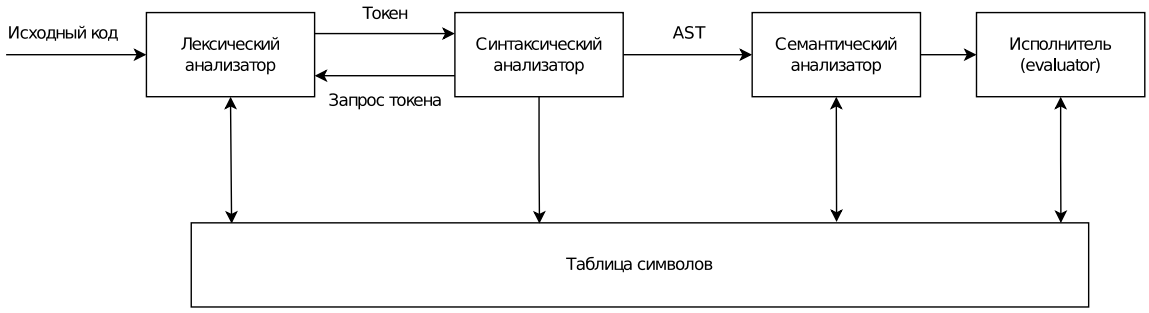

Рисунок 1 – Обобщенная структура интерпретатора

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 и работы с ранее вычисленными значениями выражений. В качестве примера можно рассмотреть следующий код:

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

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 2.

В качестве примера работы алгоритма на рисунках 3-6 приведены схемы алгоритмов семантического анализа некоторых выражений.

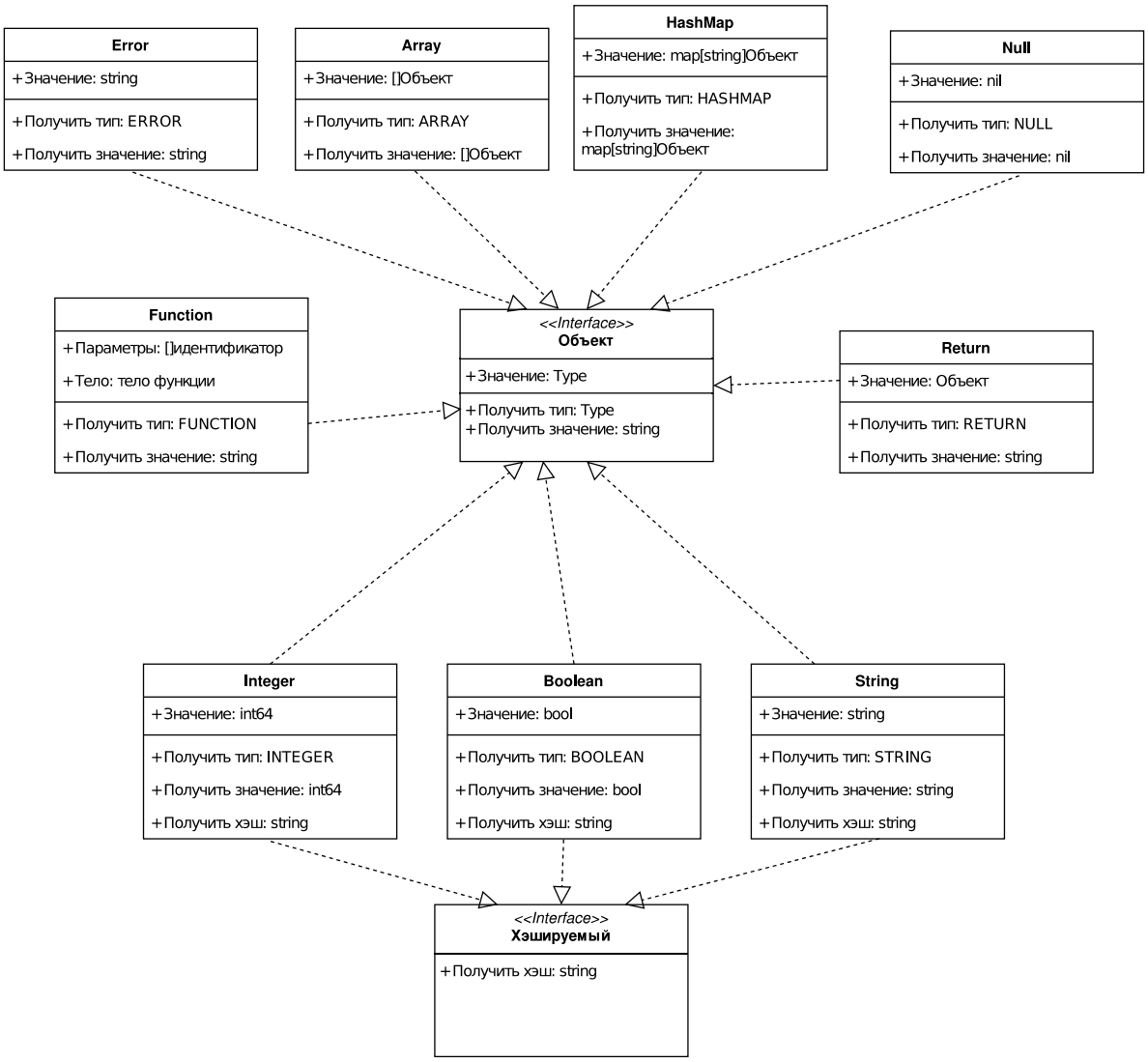

Рисунок 2 – Диаграмма классов

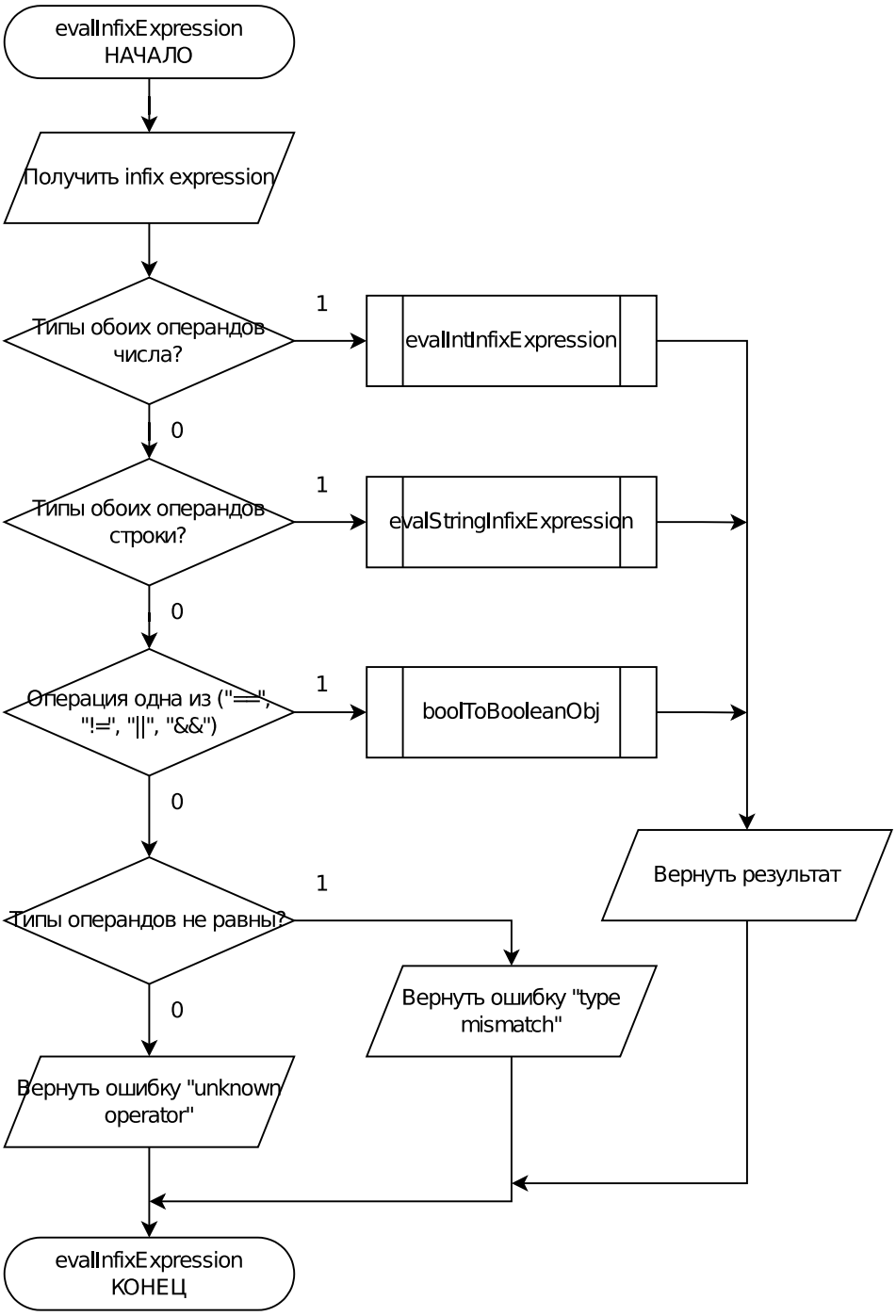

Рисунок 3 – Схема алгоритма «evalInfixExpression»

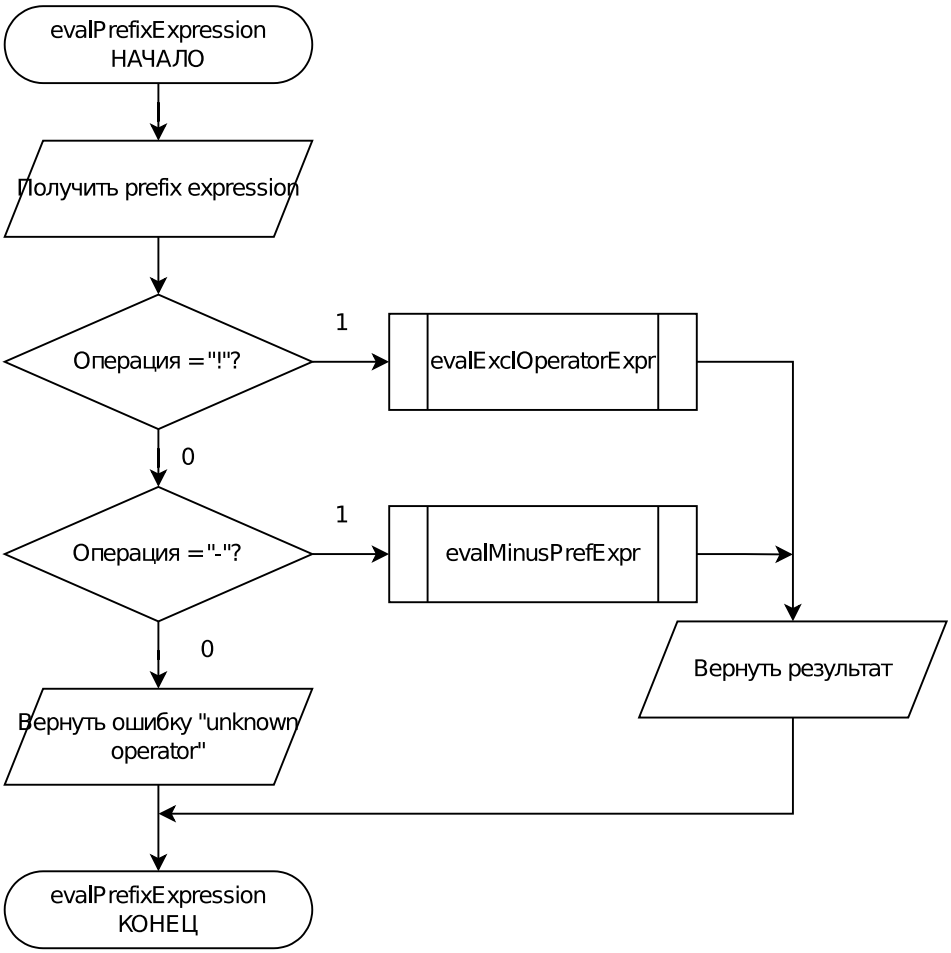

Рисунок 4 – Схема алгоритма «evalPrefixExpression»

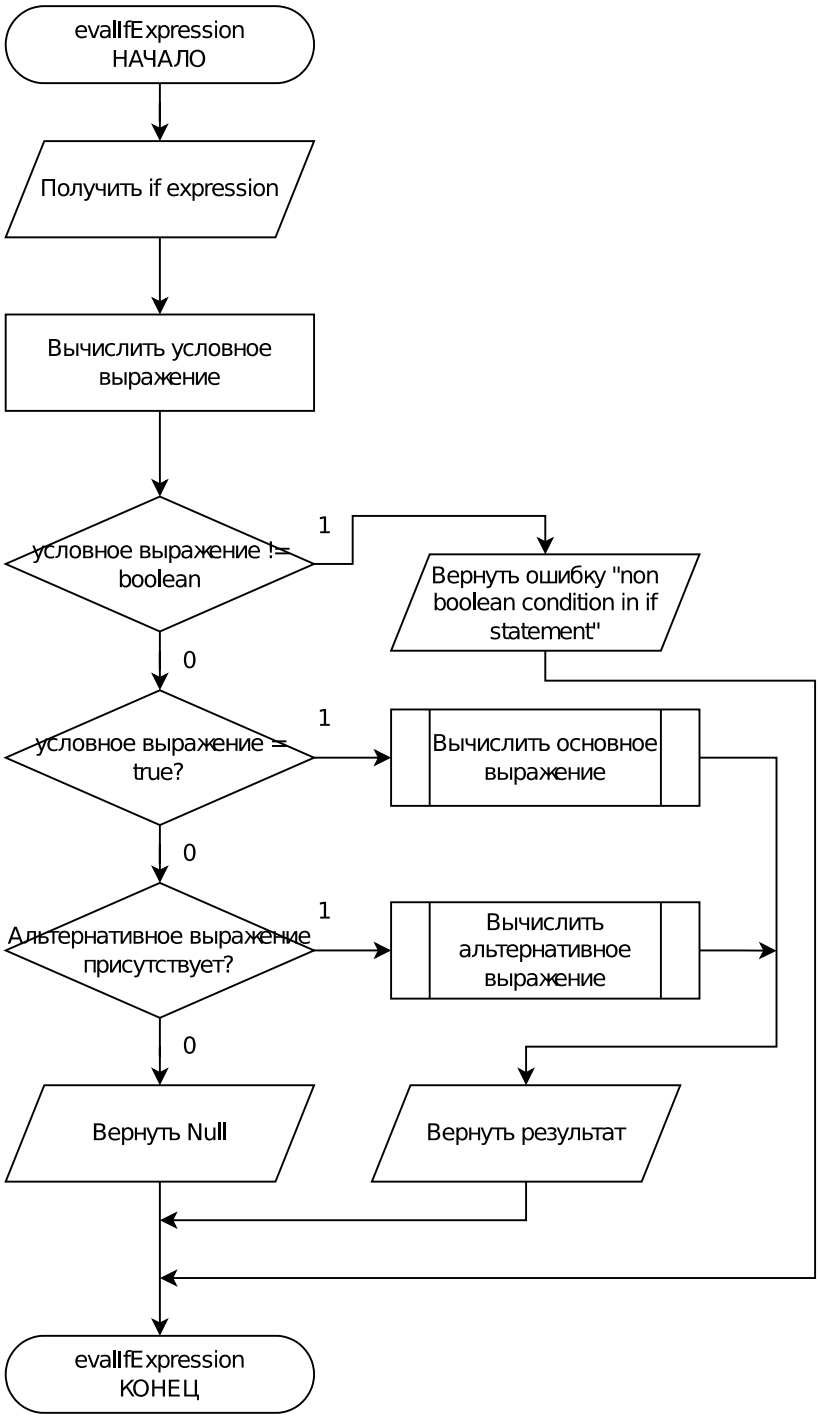

Рисунок 5 - Схема алгоритма «evalIfExpression»

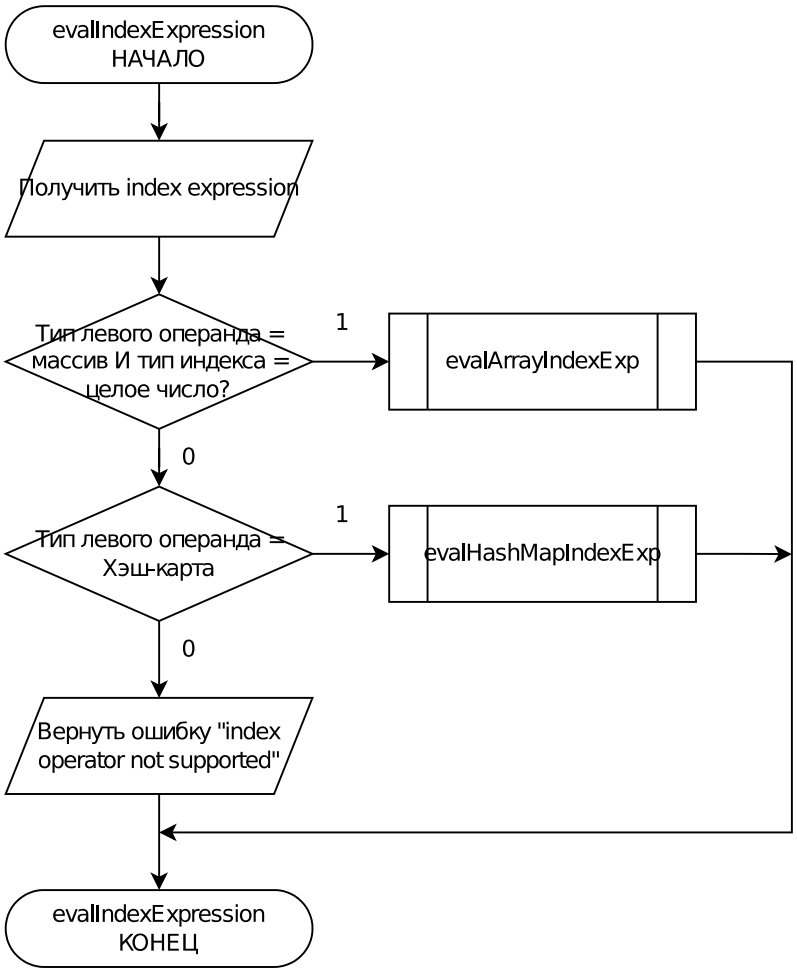

Рисунок 6 - Схема алгоритма «evalIndexExpression»

3.1.2 Программная реализация

Программная реализация этого проекта выполнена на языке программирования Go.

Каждое вычисленное значение выражения представляется в виде структуры, которая соответствует некоторому «объекту» интерфейсного типа:

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

```
}
```

Ниже приведен пример структуры, которая представляет данные типа «целое число»:

```
type Integer struct {
    Value int64
}

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

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.

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 программы.

Полный код реализации объектной системы представлен в приложении А.

Семантический анализ выполняется во время рекурсивного прохода по узлам AST и генерирует ошибку в случае её обнаружения. В противном случае начинается вычисление значение выражения.

Фрагмент кода функции разбора инфиксного выражения представлен на рисунке 7.

```
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 evalIntInfixExpression(op, left, right)
	case left.Type() == object.STRING_OBJ && right.Type() == object.STRING_OBJ:
		return evalStringInfixExpression(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())
	}
}
```

Рисунок 7 - Фрагмент кода функции разбора инфиксного выражения

Фрагменты кода семантического анализатора находятся в приложении Б.

3.2 Разработка исполнителя

В данном разделе необходимо выполнить разработку алгоритмов функционирования и программную реализацию исполнителя программного кода предметно-ориентированного языка.

3.2.1 Разработка алгоритмов функционирования

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 виде интерпретатора 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.

Данный этап 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, 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 получено из окружения.

В качестве примера работы алгоритма на рисунках 8-10 приведены схемы алгоритма вычисления некоторых выражений.

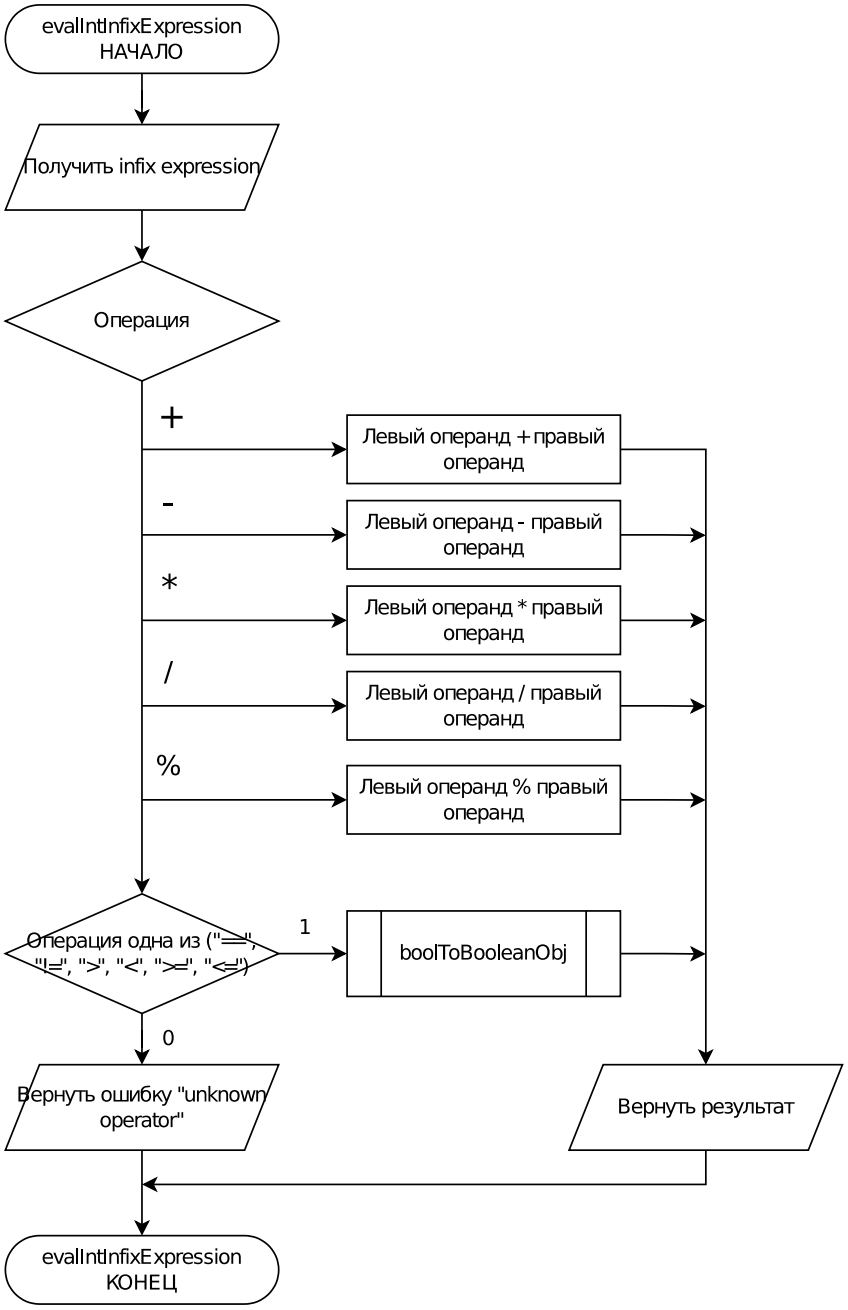

Рисунок 8 – Схема алгоритма вычисления целочисленного инфиксного выражения

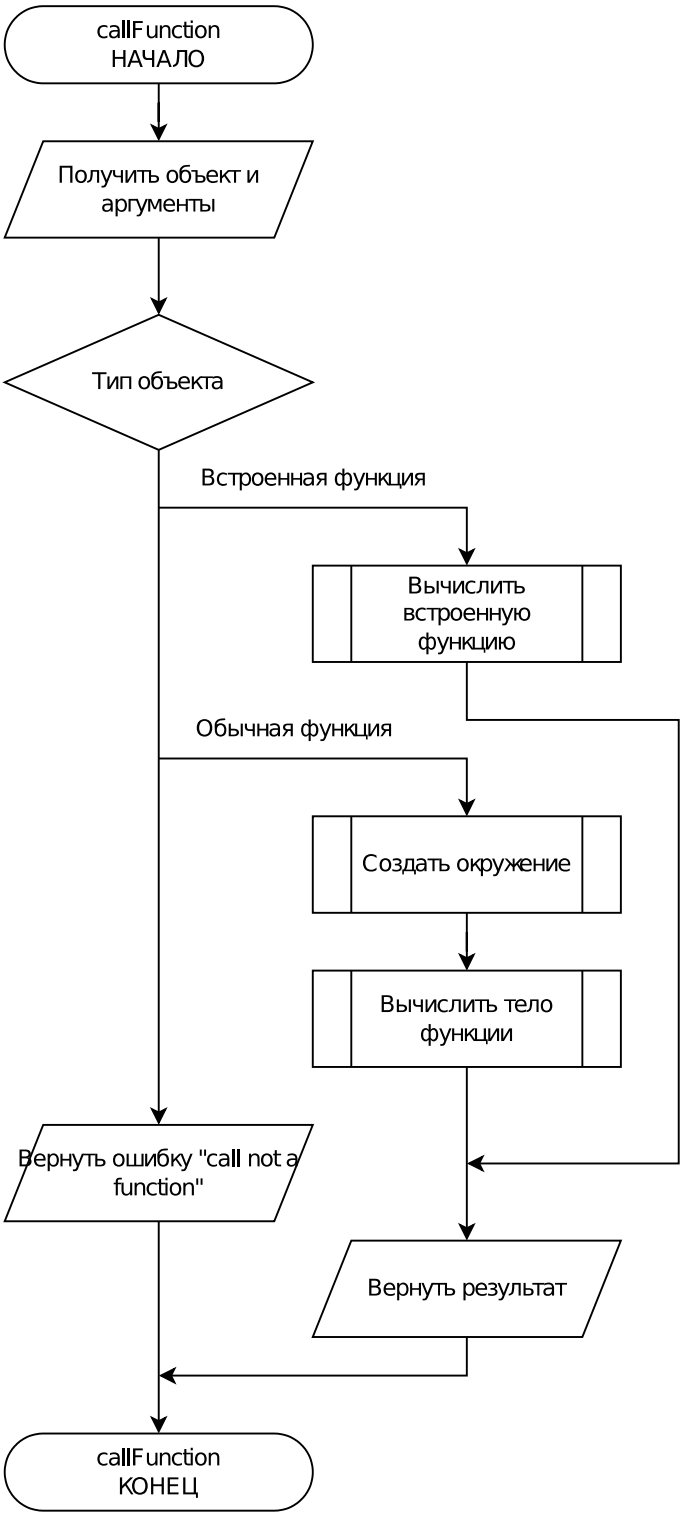

Рисунок 9 – Схема алгоритма вычисления вызова функции

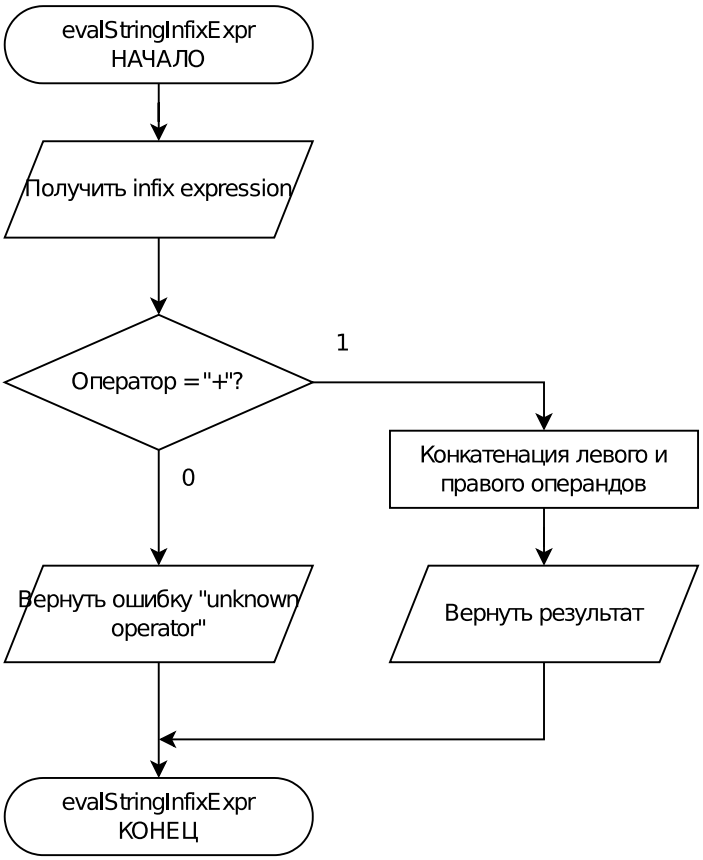

Рисунок 10 – Схема алгоритма вычисления инфиксного выражения для строк

3.2.2 Программная реализация

В общем виде процесс выполнения 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. Некоторые составляющие объектной системы описаны в п. 3.1.2.

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. Ниже приведен пример реализации данной структуры:

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

За обработку узлов AST отвечает единственная функция, которая определяет тип узла и передает управление соответствующей функции, которая 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 11.

```
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 11 – Фрагмент кода функции получения и обработки узлов AST

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 приведен на рисунке 12.

Код функции получения значения переменной из окружения приведен на рисунке 13.

```
func (e *Env) Set(key string, val Object) Object {
	e.store[key] = val
	return val
}
```

Рисунок 12 – Исходный код функции записи переменной в окружение

```
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
}
```

Рисунок 13 – Исходный код функции получения значения переменной из окружения

В аргументах функции могут быть определены параметры, имена которых соответствуют объявленным ранее переменным. Это некорректное поведение, при котором первое объявленное значение будет перезаписано другим. Пример кода такой ситуации приведен на рисунке 14.

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

Рисунок 14 – Пример кода некорректного поведения

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 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.

Код реализации встроенных функций приведен в приложении Г.

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. Ниже приведен наглядный пример:

X = { “name”: “Bob” }

X[“name”]

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 15.

```
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 15 – Фрагмент кода реализации хэш-карт

Фрагмент кода функции вычисления целочисленного инфиксного выражения приведен на рисунке 16.

```
func evalIntInfixExpression(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 16 – Фрагмент кода функции вычисления целочисленного инфиксного выражения

Фрагменты кода исполнителя приведены в приложении В.

Пример успешного выполнения программы для указанных входных данных представлен на рисунке 17.

Входная строка: 5 + 1 * 20 / (5 + 5)

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

Рисунок 17 – Пример успешного выполнения программы

Пример завершения работы программы с семантической ошибкой для указанных ниже входных данных показан на рисунке 18.

Входные данные:

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

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

Рисунок 18 – Пример завершения работы программы с семантической ошибкой

3.3 Тестирование

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

Выделяют три основных вида тестирования:

1) Модульное тестирование

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

2) 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.

3) 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 семантического анализатора и исполнителя выполнена с помощью модульных тестов. Написаны тесты для большинства функций. Тест-кейсы приведены в таблицах 1-13. Фрагменты кода тестирования представлены в приложении Д.

В ходе запуска тестирования все тесты выполнились успешно. Результат тестирования представлен на рисунке 19.

```
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 19 – Результаты запуска тестов

Таблица 1 – 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 2 – 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 3 – 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 4 – Тест-кейсы исполнения выражения return

Входные данные	Ожидаемый результат
return 15	15
return 25; 1;	25
if (true) { return 99 }	99

if (1 > 0) {
if (2 > 0) {
return 3
}
return 1;
}
	3

if (0 == 0) {
if (2 == 0) {
return 3
}
return 1;
}
	1

Таблица 5 – Тест-кейсы исполнения выражения присваивания

Входные данные	Ожидаемый результат
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

Таблица 6 – Тест-кейсы исполнения вызова функций

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 7 – Тест-кейсы исполнения строковых выражений

Входные данные	Ожидаемый результат
"Hello Earth"	Hello Earth
"Hello" + " " + "Earth"	Hello Earth

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 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 10 – Тест-кейсы исполнения хэш-карт

Входные данные	Ожидаемый результат

x = "v";
{
"a": 1,
"bb": 10*10,
x: 4,
"qq"+"ww": 123,
true: 1,
false: 0
}
	{
	"a": 1,
	"bb": 100,
	"v": 4,
	"qqww": 123,
	true: 1,
	false: 0
	}

Таблица 11 – 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

Продолжение таблицы 11

Входные данные	Ожидаемый результат
{false: -1}[false]	-1
y = "key"; {"key": 0}[y]	0

Таблица 12 – 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
len([1, 2])	2
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
last([3, 2, 1])	1
last("a")	argument must be ARRAY, got: STRING

Таблица 13 – 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

Заключение

В ходе выполнения курсового проекта по разработке модуля интерпретации предметно-ориентированного языка были проведены анализ предметной области, алгоритмы функционирования семантического анализатора и исполнителя, осуществлена их программная реализация и тестирование.

В результате выполнения курсового проекта был разработан модуль интерпретации предметно-ориентированного языка.

В качестве направления дальнейшего развития можно рассмотреть выполнение интеграции с конструктором Telegram ботов.

Приложение А

(обязательное)

Листинга 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(f.Body.ToString())

	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" }
```

Приложение Б

(обязательное)

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())
		}
}
```

Приложение В

(Обязательное)

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
}
```

Приложение Г

(Обязательное)

Листинг кода реализации встроенных функций

```
package evaluator

import (
	"github.com/botscubes/bql/internal/object"
)

var builtins = map[string]*object.Builtin{
	"len": {
		Fn: func(args ...object.Object) object.Object {
			if len(args) != 1 {
				return newError("wrong number of arguments: %d want: 1", len(args))
			}

			switch arg := args[0].(type) {
			case *object.String:
				return &object.Integer{Value: int64(len(arg.Value))}
			case *object.Array:
				return &object.Integer{Value: int64(len(arg.Elements))}
			default:
				return newError("type of argument not supported: %s", arg.Type())
			}
		},
	},
	"push": {
		Fn: func(args ...object.Object) object.Object {
			if len(args) != 2 {
				return newError("wrong number of arguments: %d want: 2", len(args))
			}

			if args[0].Type() != object.ARRAY_OBJ {
				return newError("first argument must be ARRAY, got: %s", args[0].Type())
			}

			args[0].(*object.Array).Elements = append(args[0].(*object.Array).Elements, args[1])
			return args[0]
		},
	},
	"first": {
		Fn: func(args ...object.Object) object.Object {
			if len(args) != 1 {
				return newError("wrong number of arguments: %d want: 1", len(args))
			}

			if args[0].Type() != object.ARRAY_OBJ {
				return newError("argument must be ARRAY, got: %s", args[0].Type())
			}

			arr := args[0].(*object.Array)
			if len(arr.Elements) > 0 {
				return arr.Elements[0]
			}

			return NULL
```

```
		},
	},
	"last": {
		Fn: func(args ...object.Object) object.Object {
			if len(args) != 1 {
				return newError("wrong number of arguments: %d want: 1", len(args))
			}

			if args[0].Type() != object.ARRAY_OBJ {
				return newError("argument must be ARRAY, got: %s", args[0].Type())
			}

			arr := args[0].(*object.Array)
			if len(arr.Elements) > 0 {
				return arr.Elements[len(arr.Elements)-1]
			}

			return NULL
		},
	},
}
```

Приложение Д

(Обязательное)

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 Е

(Обязательное)

Описание формальной грамматики

```
Program = Statement+

Statement = AssignStmt | FunctionDecl | ExpressionStmt | ReturnStmt | BlockStmt | IfStmt .
ExpressionStmt = Expression .
Identifier = (letter | "_") { letter | "_" | digit } .

Expression = UnaryExpr | Expression binary_op Expression .
UnaryExpr = PrimaryExpr | unary_op UnaryExpr .

PrimaryExpr = Operand | PrimaryExpr Index | CallExpr .
Index = "["Expression "]" .

AssignStmt = Identifier assign_op Expression .
ReturnStmt = "return" [Expression] .
BlockStmt = "{" StatementList "}" .
StatementList = { Statement ";" } .

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

FunctionDecl = "fn(" [ParameterList] ")" BlockStmt .
ParameterList = Identifier { "," Identifier } .
Arguments = "(" [ ExpressionList ] ")" .

CallExpr = Identifier Arguments .
ExpressionList = Expression { "," Expression } .

Array = "[" [ ExpressionList ] "]" .

Key = stringLiteral | intLiteral | Identifier | Expression .
KeyedElement  = [ Key ":" Expression ] .
Map = "{" KeyedElement { "," KeyedElement } "}" .

Operand = Literal | "(" Expression ")" .
Literal = intLiteral | stringLiteral | Array | Map .
intLiteral = digit { digit } .
stringLiteral = `"` { ascii_char } `"` .

binary_op = "||" | "&&" | rel_op | add_op | mul_op .
rel_op = "==" | "!=" | "<" | "<=" | ">" | ">=" .
add_op = "+" | "-" .
mul_op = "*" | "/" | "%" .
assign_op = "=" .

unary_op = "-" | "!" .

digit = "0" ... "9" .
letter = "A" ... "Z" | "a" ... "z" .
ascii_char = (* ascii character *) .
```

Начальное состояние, с которого начинается разбор – Program.

Приложение Ж

(Обязательное)

Перечень сокращений

РБНФ – расширенная Бэкуса-Наура форма

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

AST - abstract syntax tree (абстрактное синтаксическое дерево)

Приложение З

(справочное)

Библиографический список

1. Расширенная форма Бэкуса – Наура – Википедия [Электронный ресурс]. – Режим доступа: https://ru.wikipedia.org/wiki/Расширенная_форма_Бэкуса_—_Наура.
2. Ахо, Альфред. Компиляторы: принципы, технологии и инструментарий, “И.Д. Вильямс”, 2003 – 768 с.
3. Documentation – The Go Programming Language [Электронный ресурс]. – Режим доступа: https://go.dev/doc/.
4. Предметно-ориентированный язык — Википедия [Электронный ресурс]. – Режим доступа: https://ru.wikipedia.org/wiki/Предметно-ориентированный_язык.