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.)
«___» _____2023г.

Разработка лексического и синтаксического анализатора для 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 2023

Реферат

Крючков И. С. Разработка лексического и синтаксического анализатора для предметно-ориентированного языка. ТПЖА.090301.331 ПЗ: Курс. проект / ВятГУ, каф. ЭВМ; рук. Долженкова М. Л. - Киров, 2023. – ПЗ 62 с, 19 рис., 2 табл., 5 источников, 7 прил.

ПРЕДМЕТНО-ОРИЕНТИРОВАННЫЙ ЯЗЫК, ГРАММАТИКА, ЛЕКСИЧЕСКИЙ АНАЛИЗ, СИНТАКСИЧЕСКИЙ АНАЛИЗ, ТОКЕН, КОНСТРУКТОР, GO.

Объект курсового проекта – предметно-ориентированный язык.

Предмет - лексический и синтаксический анализатор.

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

Результат работы – разработанная грамматика, лексический и синтаксический анализаторы предметно-ориентированного языка расширяющего функциональные возможности конструктора Telegram ботов.

Содержание

Введение

В современном мире стали популярными такие приложения для быстрого общения как мессенджеры. Таких приложений достаточно много, но большинство пользователей сети интернет все чаще отдают предпочтение мессенджеру Telegram как наиболее удобному и надежному. Однако, для создания и управления Telegram ботами требуется определенный уровень технических знаний и навыков программирования, что может быть преградой для многих пользователей.

Конструктор позволяет широкому кругу пользователей создавать Telegram ботов с помощью визуального редактора. Однако возможности визуального редактора ограничены и их возможностей зачастую недостаточно для построения сложных, специфичных ботов. Расширение возможностей платформы возможно за счёт использования предметно-ориентированного языка программирования для конструктора Telegram ботов. С помощью написания инструкцию на данном языке, пользователи конструктора могут гибко задавать логику работы ботов.

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 конструктора Telegram ботов.

1.1 Актуальность темы

Боты в мессенджере Telegram становятся все более популярными и число их пользователей постоянно растет. Они помогают пользователям выполнять типичные рутинные действия в автоматизированном режиме, значительно упрощая им жизнь. Для владельцев же самих ботов они стали незаменимыми помощниками в работе.

Telegram-боты имеют множество плюсов, таких как:

– Круглосуточный доступ;

– Моментальный ответ на запрос пользователя;

– Удобство использования, интуитивно понятный интерфейс;

– Не требуют установки дополнительных программ, общение с ботом ведется через мессенджер;

– Широкий набор реализуемых функций

Telegram-бот используют в коммерческой деятельности для следующих сфер и задач:

– Развлечения;

– Поиск и обмен файлами;

– Предоставление новостей;

– Утилиты и инструменты;

– Интеграция с другими сервисами;

– Осуществление онлайн-платежей.

С популярностью ботов стали появляться все больше различных конструкторов, которые позволяют без наличия специальных знаний и навыков создать своего бота всего в несколько кликов.

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 конструктора.

Первое предназначение – упрощение работы. Ведь не все обладают знаниями и навыками программирования. Когда боты только появились, их могли разрабатывать только программисты, обладающие соответствующим опытом и навыками

Помимо того, что конструкторы позволяют расширить аудиторию, способную создавать Telegram-ботов, они экономят время разработчикам. При наличии конструктора нет необходимости разрабатывать каждый раз отдельное приложение для выполнения типовых задач, так как конструктор предоставляет необходимый набор инструментов для быстрого создания бота.

В ранее выполненных курсовых проектах была спроектирована структура и разработан прототип визуального конструктора Telegram ботов. Разработанный прототип позволяет пользователям создавать ботов с помощью реализованных в системе компонентов и запускать их.

Несмотря на все преимущества, у визуальных конструкторов есть значительные ограничения, которые не позволяют гибко настраивать логику работы бота. 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 описывать логику работы бота с использованием команд и выражений, что придаёт большую гибкость при создании сценариев функционирования бота.

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.

1.2 Постановка задачи

1.2.1 Основание для разработки

Предметно-ориентированный язык конструктора Telegram ботов 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 конструктора Telegram ботов.

Задачи:

– разработать формальную грамматику предметно-ориентированного языка;

– разработать программу, включающую в себя лексический анализатор для разбора входного кода на токены и синтаксический анализатор для построения абстрактного синтаксического дерева и проверки корректности синтаксических конструкций языка.

1.2.3 Определение ключевых конструкция языка

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

Поддерживаемые типы данных: строки, числа, булевы значения.

Кроме этого, язык должен поддерживать составной тип данных – массив.

Набор основных операций языка:

- арифметические операции: сложение, вычитание, умножение, деление, операция нахождение остатка от деления;
- операции сравнения;
- вызов функций.

Вывод

В данном разделе был проведен анализ предметной области и основываясь на ранее выполненных курсовых проектах по проектированию и разработке конструктора Telegram ботов, определены основные требования к разрабатываемому программному продукту.

Telegram боты являются функциональными инструментами для многих пользователей, однако для их разработки зачастую требуются навыки программирования, что усложняет их внедрение в бизнес-процессы компаний. Визуальный конструктор Telegram ботов значительно упрощает процесс создания и запуска ботов, однако, он имеет функциональные ограничения, которые препятствуют разработке ботов со сложной логикой работы. Предметно-ориентированный язык для конструктора Telegram ботов позволяет расширить возможности визуального конструктора и создавать уникальных ботов, в соответствии с заданными потребностями.

Таким образом, проблема является актуальной, и разработка лексического и синтаксического анализатора для предметно-ориентированного языка является актуальной.

2 Разработка грамматики

В данном разделе проводится разработка формальной грамматики языка.

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.

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:

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 } .

Operand = Literal | "(" Expression ")" .
Literal = intLiteral | stringLiteral
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.

Вывод

В данном разделе была разработана и описана с помощью расширенной формы Бэкуса-Наура формальная грамматика предметно-ориентированного языка.

3 Разработка анализаторов

На данном этапе работы необходимо в соответствие с поставленными требованиями разработать ключевые компоненты транслятора языка – лексический и синтаксический анализаторы.

3.1 Разработка лексического анализатора

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

3.1.1 Разработка алгоритмов функционирования

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.

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

- идентификаторы;
- числа;

– строки;
– разделители;
– операторы (арифметические, сравнения и т.д);
– скобки;
– специальные:
– ключевые слова.

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

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	>

Продолжение таблицы 1.

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

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.

Схема взаимодействия лексического и синтаксического анализаторов показано на рисунке 1.

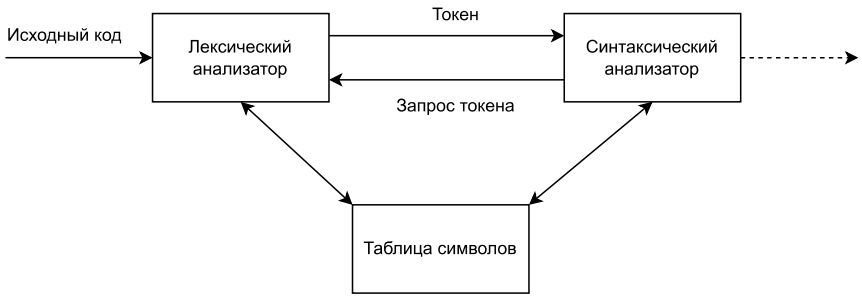

Рисунок 1 – Схема взаимодействия лексического и синтаксического анализаторов

При получении запроса на следующий токен 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 2 показана диаграмма для определения токенов «=» 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».

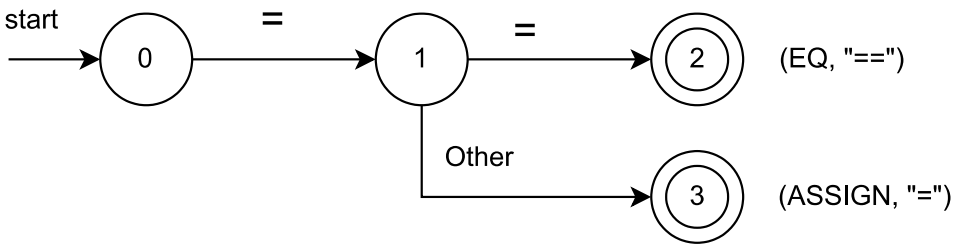

Рисунок 2 – Диаграмма переходов для определения «=» и «==»

Диаграмма для распознавания целого числа представлена на рисунке 3.

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.

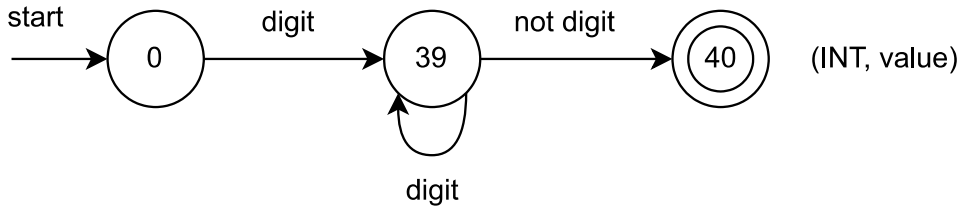

Рисунок 3 – Диаграмма переходов для определения целого числа

Считывание ключевых слов и идентификаторов показано на диаграмме передов на рисунке 4. Из начального состояния происходит переход в состояние 39, 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.

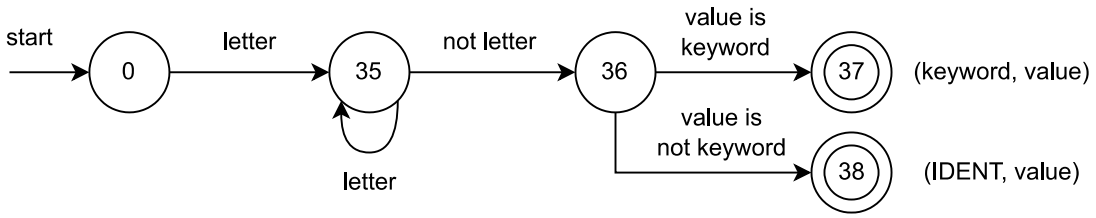

Рисунок 4 – Диаграмма переходов для определения идентификаторов и ключевых слов

Полная диаграмма переходов состояний представлена на рисунке 5.

Процесс распознавания токена начинается с начального состояния 0. В зависимости 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 конченом 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.

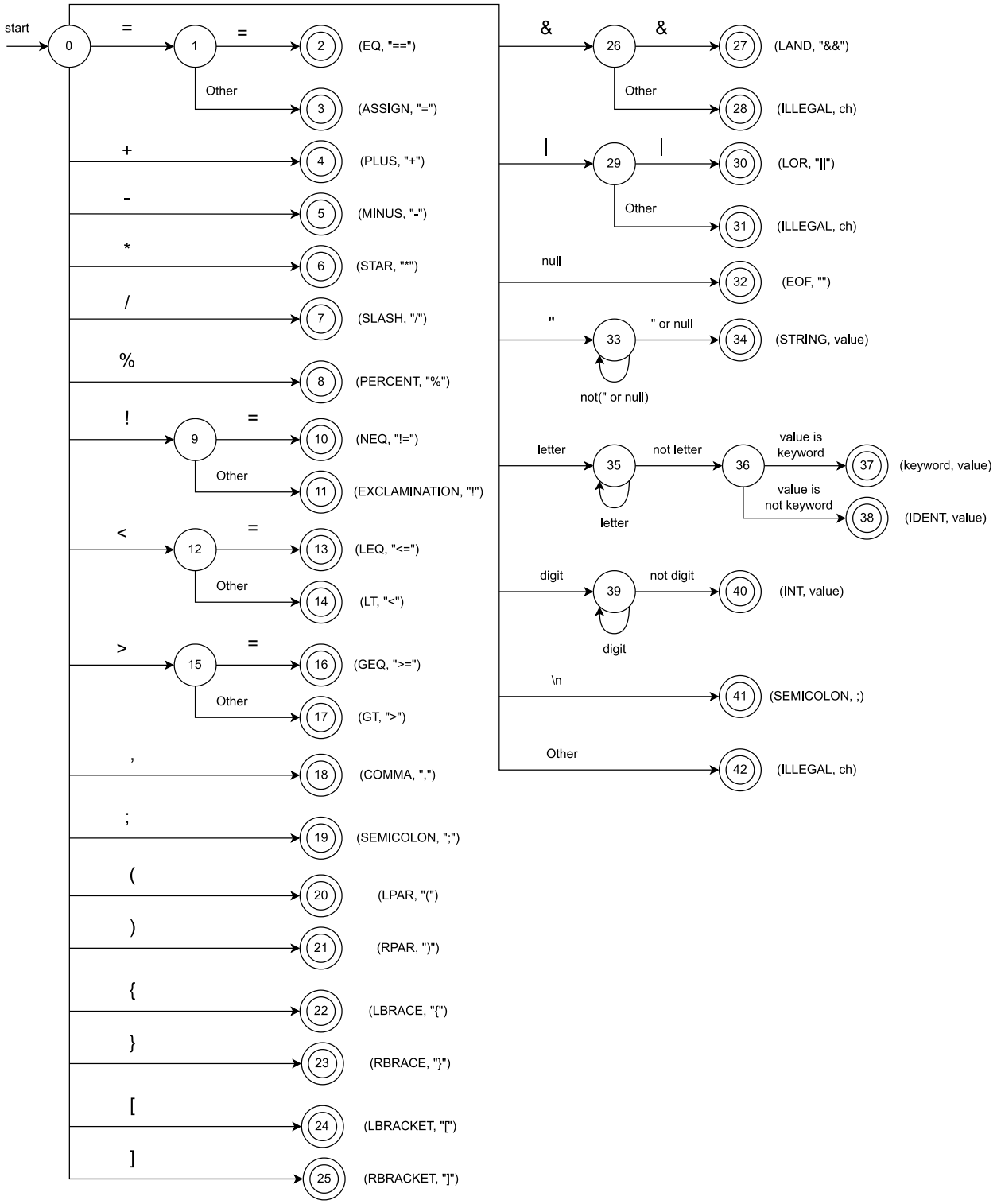

Рисунок 5 – Полная диаграмма переходов состояний

3.1.2 Программная реализация

Программная реализация этого проекта выполнена на языке программирования Go.

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, реализованную следующим образом:

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

Некоторые из типов токенов:

```
IDENT  = "IDENT"  // x, t, add
INT    = "INT"    // 123
STRING = "STRING" // "abcde"
ASSIGN       = "="
PLUS         = "+"
MINUS        = "-"
STAR         = "*"
```

Полный код реализации токенов представлен в приложении А.

Лексер реализован в виде конструкции switch-case. Фрагмент кода функции определения токена:

```
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)
```

Полный код лексического анализатора находится в приложении Б.

3.2 Разработка синтаксического анализатора

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

3.2.1 Разработка алгоритмов функционирования

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.

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.

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 описан 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	(
9	[

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

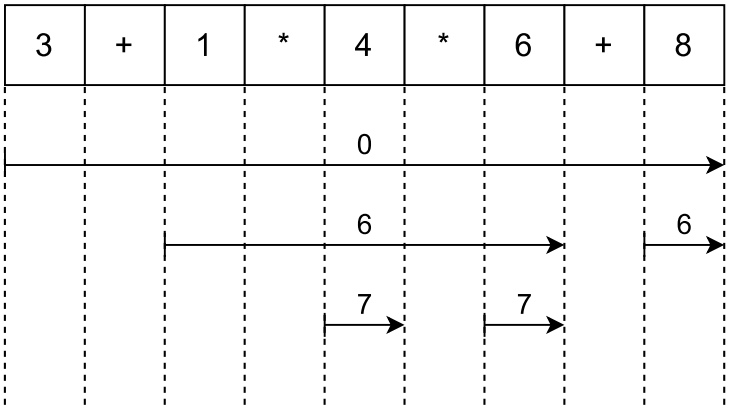

Рисунок 6 – Диаграмма рекурсивных вызовов

Абстрактное синтаксическое дерево для данного примера приведено на рисунке 7.

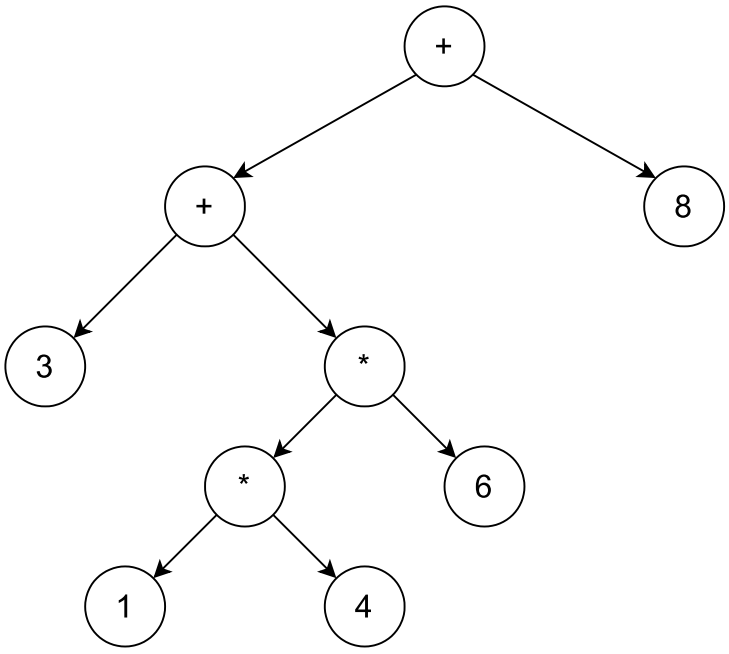

Рисунок 7 – Абстрактного синтаксическое дерево

Некоторые схемы алгоритма построения абстрактного синтаксического дерева представлены на рисунках 8-15.

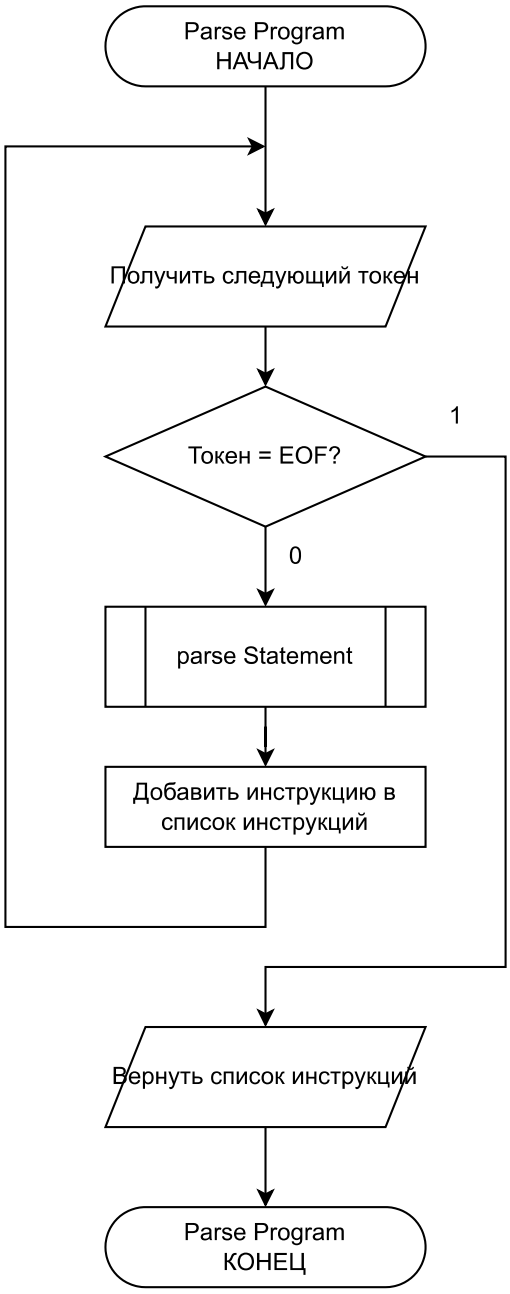

Рисунок 8 – Схема алгоритма «Parse Program»

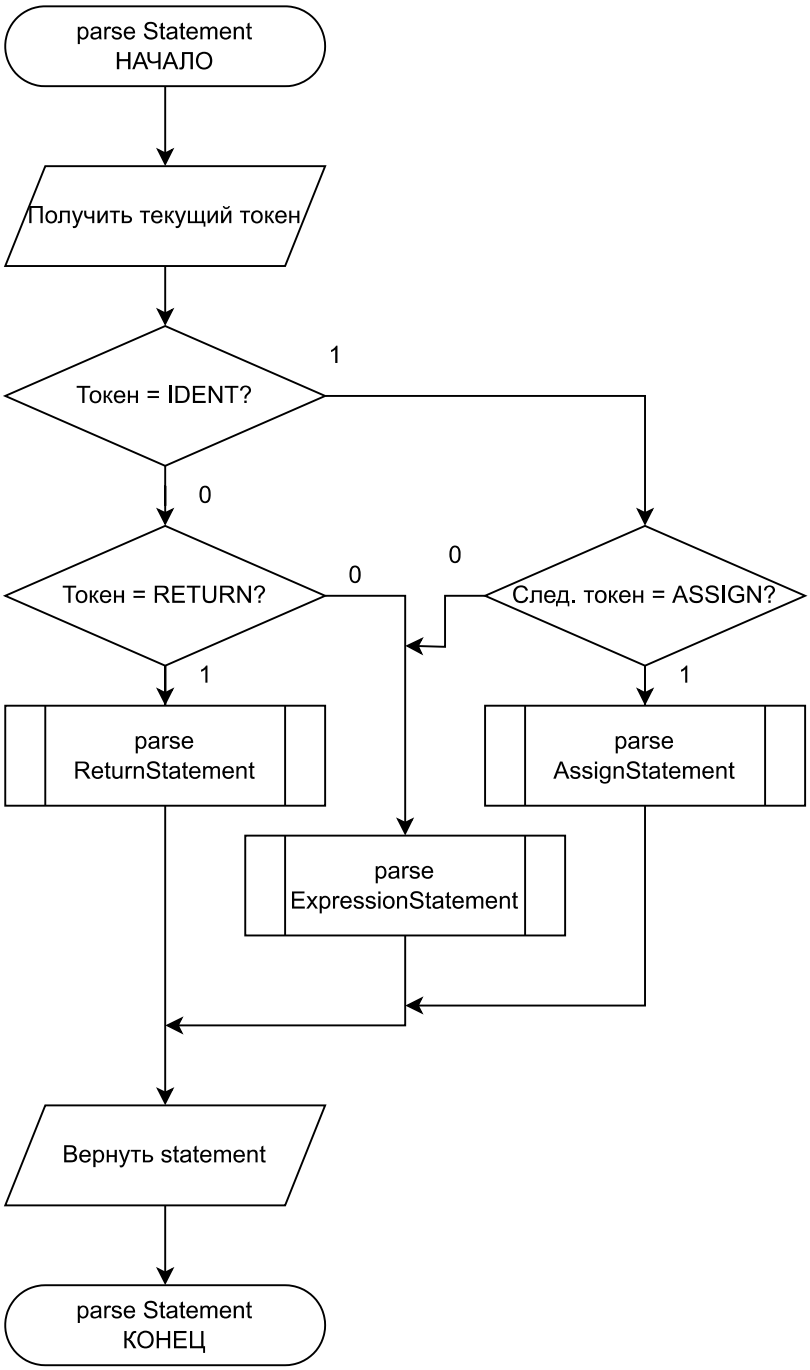

Рисунок 9 – Схема алгоритма «parse Statement»

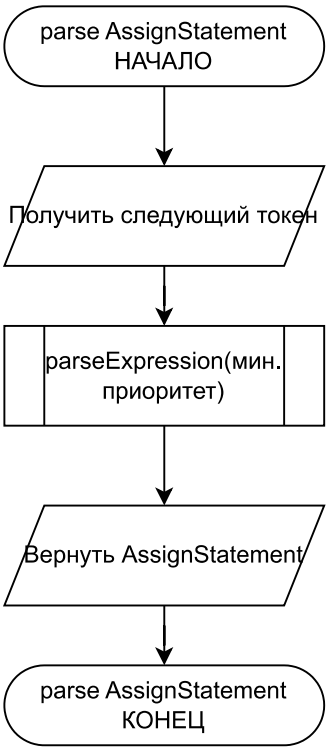

Рисунок 10 – Схема алгоритма «parse AssignStatement»

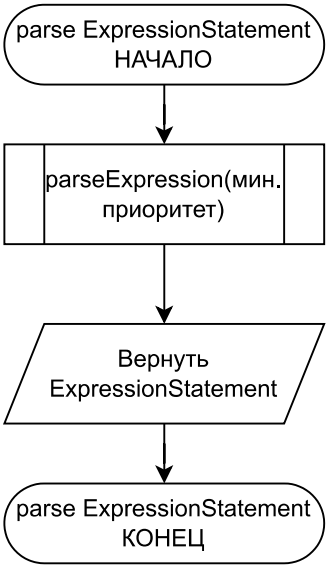

Рисунок 11 – Схема алгоритма «parse ExpressionStatement»

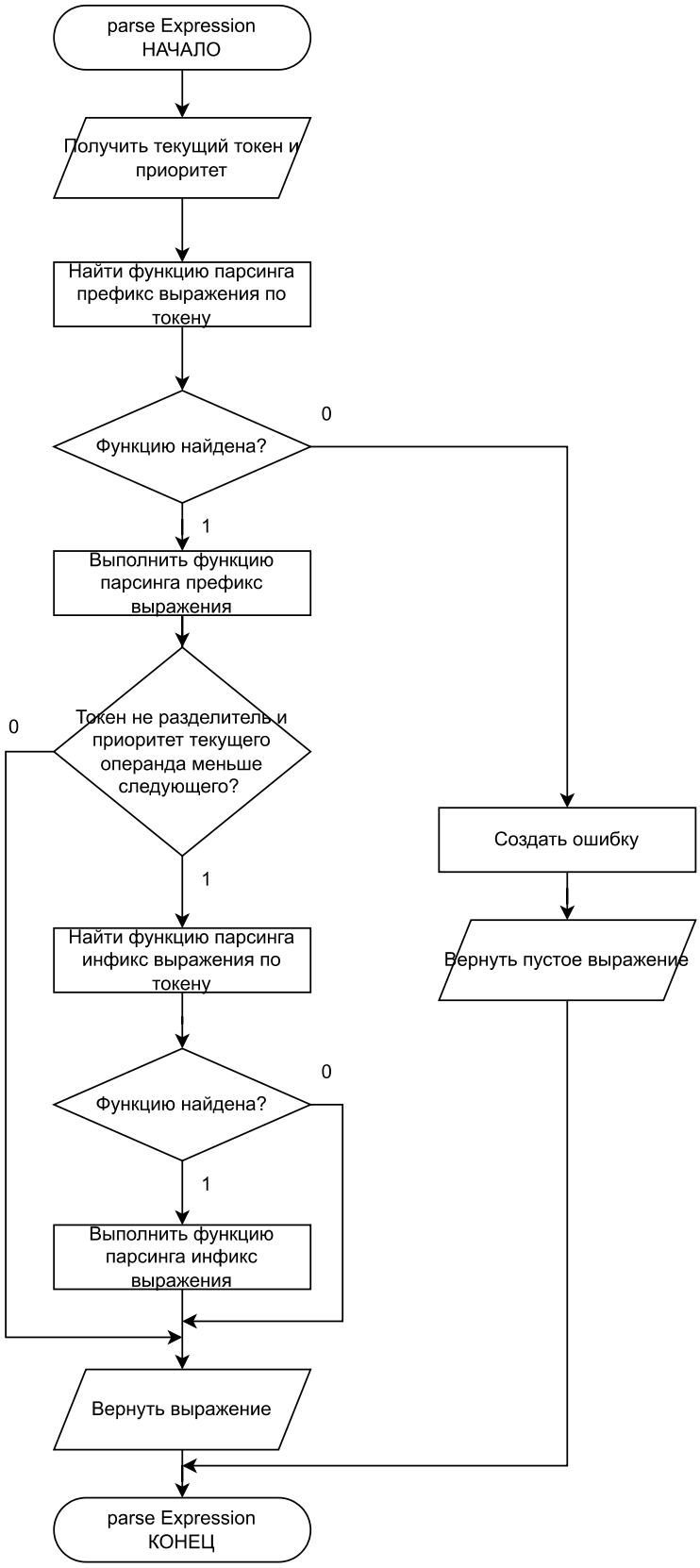

Рисунок 12 – Схема алгоритма «parse Expression»

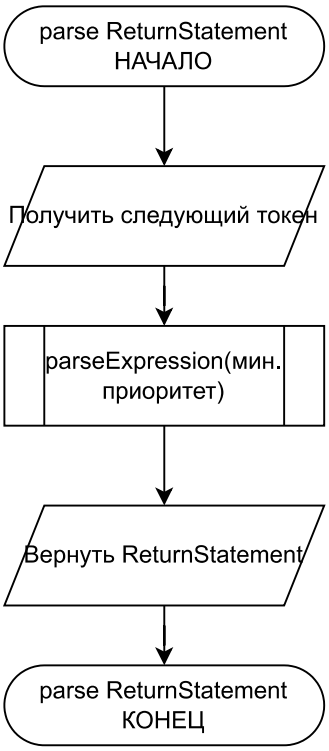

Рисунок 13 – Схема алгоритма «parse ReturnStatement»

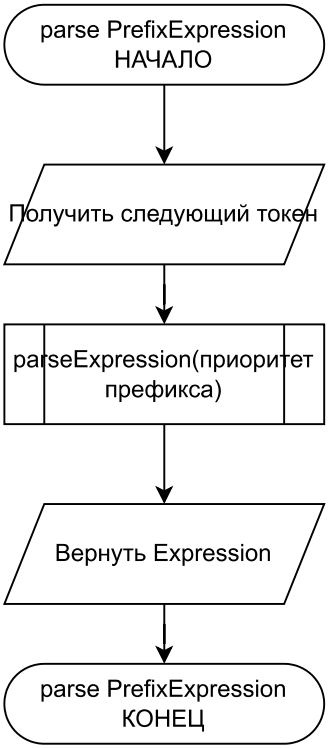

Рисунок 14 – Схема алгоритма «parse PrefixExpression»

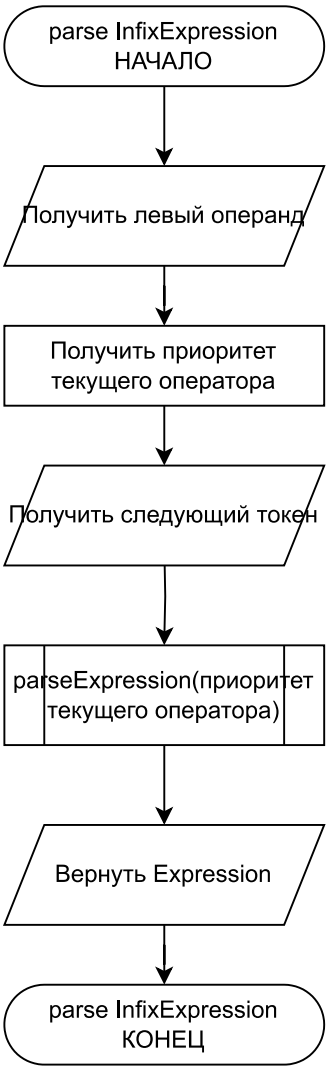

Рисунок 15 – Схема алгоритма «parse InfixExpression»

3.2.2 Программная реализация

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

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 представлена в виде интерфейсов:

```
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()
}
```

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:

```
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()
}
```

Пример кода структуры integerLineral реализующего интерфейс Expression:

```
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 }
```

Фрагменты кода синтаксического анализатора приведены в приложении В.

Пример абстрактного синтаксического дерева, полученного в результате работы программы для указанных входных данных представлен на рисунке 18. На рисунке 17 представлено графическое представление данного дерева.

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 16.

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

Рисунок 16 – Результат работы синтаксического анализатора в виде строки

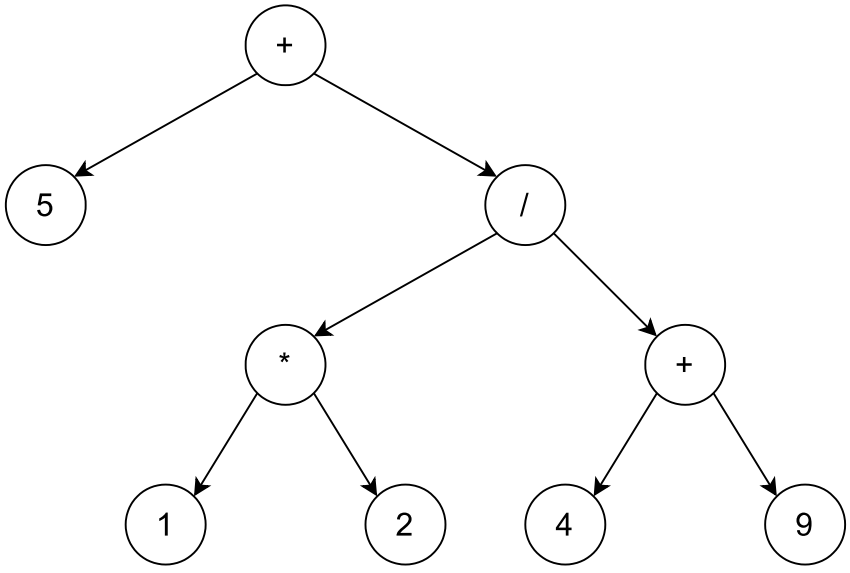

Рисунок 17 – AST для указанной входной строки

```
(*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 18 – Результат работы синтаксического анализатора в виде AST

3.3 Тестирование

Тестирование является один 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 лексического и синтаксического анализаторов выполнена с помощью модульных тестов. Написаны тесты для всех большинства функций пакетов лексического и синтаксического анализаторов. Исходный код с тестами лексического анализатора находится в приложении Г. Фрагменты кода тестирования синтаксического анализатора представлен 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/lexer 0.121s
ok      github.com/botscubes/bql/internal/parser        0.025s
```

Рисунок 19 – Результаты запуска тестов

Заключение

В ходе выполнения курсового проекта по разработке лексического и синтаксического анализатора для предметно-ориентированного языка были проведены анализ предметной области, разработана грамматика предметно-ориентированного языка, алгоритмы функционирования лексического и синтаксического анализаторов, осуществлена их программная реализация и тестирование.

В результате выполнения курсового проекта был разработан лексический и синтаксический анализатор для предметно-ориентированного языка.

В качестве направления дальнейшего развития можно рассмотреть разработку семантического анализатора для проверки сематической корректности программы и доработку транслятора до полной работоспособности, что позволит преобразовывать код программы в исполняемый код. Выполнить интеграцию с конструктором Telegram ботов.

Приложение А

(обязательное)

Листинга кода объявления токенов

```
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 = ";"

	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
}
```

Приложение Б

(обязательное)

Листинг 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.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) {
                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
}
```

Приложение В

(Обязательное)

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,
	}

	// LBRACKET = "["
	// RBRACKET = "]"

	// 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

	// 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

	// 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) 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
}

func (p *Parser) parseFunctionParameters() []*ast.Ident {
	identifiers := []*ast.Ident{}

	if p.peekTokenIs(token.RPAR) {
		p.nextToken()
		return identifiers
	}

	for {
		p.nextToken()
		if !p.curTokenIs(token.IDENT) {
			p.newError(fmt.Sprintf("failed parse %q as ident", p.curToken.Literal))
			return nil
		}

		ident := &ast.Ident{Token: p.curToken, Value: p.curToken.Literal}
		identifiers = append(identifiers, ident)
		if !p.peekTokenIs(token.COMMA) {
			break
		}

		p.nextToken()
	}

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

	return identifiers
}
```

Приложение Г

(Обязательное)

Листинг кода тестов лексического анализатора

```
package lexer

import (
	"testing"

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

type ExpectedToken struct {
	expectedType    token.TokenType
	expectedLiteral string
}

func TestNextToken(t *testing.T) {
	input := `x = 2 + 3
_ 3123 - 7
if aelse
if (x == 1) {
	[ 3, 4]
} else {
	3 <= 2
}

1 != 2
9 > 8
1 < 5

!true != false

5 % 1
0/1
"abc"
"a 1 -2 yy"
2 + 3; y = 4
"ABC" "qqq"
if (true) { 1 } else { 0 }
fn(){}
fn(x){ x }

q = fn(x,y,z){
	r = x+y
	return r * z
}

a && true;
b || true;
`

	tests := []ExpectedToken{
		{token.IDENT, "x"},
		{token.ASSIGN, "="},
		{token.INT, "2"},
		{token.PLUS, "+"},
```

```
{token.INT, "3"},
{token.SEMICOLON, "\n"},
{token.IDENT, "_"},
{token.INT, "3123"},
{token.MINUS, "-"},
{token.INT, "7"},
{token.SEMICOLON, "\n"},
{token.IF, "if"},
{token.IDENT, "aelse"},
{token.SEMICOLON, "\n"},
{token.IF, "if"},
{token.LPAR, "("},
{token.IDENT, "x"},
{token.EQ, "=="},
{token.INT, "1"},
{token.RPAR, ")"},
{token.LBRACE, "{"},
{token.LBRACKET, "["},
{token.INT, "3"},
{token.COMMA, ","},
{token.INT, "4"},
{token.RBRACKET, "]"},
{token.SEMICOLON, "\n"},
{token.RBRACE, "}"},
{token.ELSE, "else"},
{token.LBRACE, "{"},
{token.INT, "3"},
{token.LEQ, "<="},
{token.INT, "2"},
{token.SEMICOLON, "\n"},
{token.RBRACE, "}"},
{token.SEMICOLON, "\n"},
{token.INT, "1"},
{token.NEQ, "!="},
{token.INT, "2"},
{token.SEMICOLON, "\n"},
{token.INT, "9"},
{token.GT, ">"},
{token.INT, "8"},
{token.SEMICOLON, "\n"},
{token.INT, "1"},
{token.LT, "<"},
{token.INT, "5"},
{token.SEMICOLON, "\n"},
{token.EXCLAMINATION, "!"},
{token.TRUE, "true"},
{token.NEQ, "!="},
{token.FALSE, "false"},
{token.SEMICOLON, "\n"},
{token.INT, "5"},
{token.PERCENT, "%"},
{token.INT, "1"},
{token.SEMICOLON, "\n"},
{token.INT, "0"},
{token.SLASH, "/"},
{token.INT, "1"},
{token.SEMICOLON, "\n"},
{token.STRING, "abc"},
{token.SEMICOLON, "\n"},
```

```
{token.STRING, "a 1 -2 yy"},
{token.SEMICOLON, "\n"},
{token.INT, "2"},
{token.PLUS, "+"},
{token.INT, "3"},
{token.SEMICOLON, ";"},
{token.IDENT, "y"},
{token.ASSIGN, "="},
{token.INT, "4"},
{token.SEMICOLON, "\n"},
{token.STRING, "ABC"},
{token.STRING, "qqq"},
{token.SEMICOLON, "\n"},
{token.IF, "if"},
{token.LPAR, "("},
{token.TRUE, "true"},
{token.RPAR, ")"},
{token.LBRACE, "{"},
{token.INT, "1"},
{token.RBRACE, "}"},
{token.ELSE, "else"},
{token.LBRACE, "{"},
{token.INT, "0"},
{token.RBRACE, "}"},
{token.SEMICOLON, "\n"},
{token.FUNC, "fn"},
{token.LPAR, "("},
{token.RPAR, ")"},
{token.LBRACE, "{"},
{token.RBRACE, "}"},
{token.SEMICOLON, "\n"},
{token.FUNC, "fn"},
{token.LPAR, "("},
{token.IDENT, "x"},
{token.RPAR, ")"},
{token.LBRACE, "{"},
{token.IDENT, "x"},
{token.RBRACE, "}"},
{token.SEMICOLON, "\n"},
{token.IDENT, "q"},
{token.ASSIGN, "="},
{token.FUNC, "fn"},
{token.LPAR, "("},
{token.IDENT, "x"},
{token.COMMA, ","},
{token.IDENT, "y"},
{token.COMMA, ","},
{token.IDENT, "z"},
{token.RPAR, ")"},
{token.LBRACE, "{"},
{token.IDENT, "r"},
{token.ASSIGN, "="},
{token.IDENT, "x"},
{token.PLUS, "+"},
{token.IDENT, "y"},
{token.SEMICOLON, "\n"},
{token.RETURN, "return"},
{token.IDENT, "r"},
{token.STAR, "*"},
```

```
                {token.IDENT, "z"},
                {token.SEMICOLON, "\n"},
                {token.RBRACE, "}"},
                {token.SEMICOLON, "\n"},
                {token.IDENT, "a"},
                {token.LAND, "&&"},
                {token.TRUE, "true"},
                {token.SEMICOLON, ";"},
                {token.IDENT, "b"},
                {token.LOR, "||"},
                {token.TRUE, "true"},
                {token.SEMICOLON, ";"},
                {token.EOF, ""},
        }

        l := New(input)

        for i, test := range tests {
                tok, _ := l.NextToken()

                if tok.Type != test.expectedType {
                        t.Fatalf("tests[%d] - tokentype wrong: expected=%q, got=%q",
                                i, test.expectedType, tok.Type)
                }

                if tok.Literal != test.expectedLiteral {
                        t.Fatalf("tests[%d] - literal wrong: expected=%q, got=%q",
                                i, test.expectedLiteral, tok.Literal)
                }
        }
}
```

Приложение Д

(Обязательное)

Фрагменты листинга кода тестов синтаксического анализатора

```
func TestParseAssignStatement(t *testing.T) {
	tests := []struct {
		input string
		ident string
		value any
	}{
		{"x = 56", "x", 56},
		{"y = x", "y", "x"},
		{"k = true", "k", true},
	}

	for _, test := range tests {
		l := lexer.New(test.input)
		p := New(l)
		result := p.ParseProgram()
		checkParserErrors(t, p)

		if len(result.Statements) != 1 {
			t.Fatalf("program has incorrect number of statements. got:%d",
				len(result.Statements))
		}

		stmt, ok := result.Statements[0].(*ast.AssignStatement)
		if !ok {
			t.Fatalf("result.Statements[0] is not ast.AssignStatement. got:%T",
				result.Statements[0])
		}

		if !testIdent(t, stmt.Name, test.ident) {
			return
		}

		if !testLiteralExpression(t, stmt.Value, test.value) {
			return
		}

	}
}

func TestParseReturnStatement(t *testing.T) {
	tests := []struct {
		input string
		value any
	}{
		{"return x", "x"},
		{"return true", true},
		{"return 4;", 4},
	}

	for _, test := range tests {
		l := lexer.New(test.input)
		p := New(l)
		result := p.ParseProgram()
```

```
		checkParserErrors(t, p)

		if len(result.Statements) != 1 {
			t.Fatalf("program has incorrect number of statements. got:%d",
				len(result.Statements))
		}

		returnStmt, ok := result.Statements[0].(*ast.ReturnStatement)
		if !ok {
			t.Fatalf("result.Statements[0] is not ast.ReturnStatement. got:%T",
				result.Statements[0])
		}

		if returnStmt.TokenLiteral() != "return" {
			t.Fatalf("returnStmt.TokenLiteral is not 'return'. got:%s",
				returnStmt.TokenLiteral())
		}

		if !testLiteralExpression(t, returnStmt.Value, test.value) {
			return
		}
	}
}

func TestParsePrefixExpression(t *testing.T) {
	tests := []struct {
		input    string
		operator string
		value    any
	}{
		{"-2", "-", 2},
		{"!1", "!", 1},
		{"!true", "!", true},
		{"!false", "!", false},
		{"!xyz", "!", "xyz"},
		{"-xyz", "-", "xyz"},
	}

	for _, test := range tests {
		l := lexer.New(test.input)
		p := New(l)
		result := p.ParseProgram()
		checkParserErrors(t, p)

		if len(result.Statements) != 1 {
			t.Fatalf("program has incorrect number of statements. got:%d",
				len(result.Statements))
		}

		stmt, ok := result.Statements[0].(*ast.ExpressionStatement)
		if !ok {
			t.Fatalf("result.Statements[0] is not ast.ExpressionStatement. got:%T",
				result.Statements[0])
		}

		expr, ok := stmt.Expression.(*ast.PrefixExpression)
		if !ok {
			t.Fatalf("stmt is not ast.PrefixExpression. got:%T", stmt.Expression)
		}
```

```
                if expr.Operator != test.operator {
                        t.Fatalf("expr.Operator is not %s. got:%T",
                                test.operator, expr.Operator)
                }

                if !testLiteralExpression(t, expr.Right, test.value) {
                        return
                }
        }
}
```

Приложение Е

(Обязательное)

Перечень сокращений

РБНФ – расширенная Бэкуса-Наура форма

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

Приложение Ё

(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. Top-Down operator precedence (Pratt) parsing [Электронный ресурс]. – Режим доступа: https://eli.thegreenplace.net/2010/01/02/top-down-operator-precedence-parsing.
5. Предметно-ориентированный язык — Википедия [Электронный ресурс]. – Режим доступа: https://ru.wikipedia.org/wiki/Предметно-ориентированный_язык.