

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ  
РОССИЙСКОЙ ФЕДЕРАЦИИ  
ПЕНЗЕНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

Кафедра «Математическое обеспечение и применение ЭВМ»

Курсовой проект  
по дисциплине  
«Теория языков программирования и методы трансляции»  
на тему «Разработка транслятора»

ПГУ 09.03.04 – 05КП201.15 ПЗ

Направление подготовки - 09.03.04 Программная инженерия

Выполнил студент: лсц - Лёвин М.В.  
Группа: 20ВП1

Руководитель:  
к.т.н., доцент Дорофеева Дорофеева О.С.

Работа защищена с оценкой отлично  
Преподаватели Дорофеева  
Дата защиты 23.12.2022

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ  
РОССИЙСКОЙ ФЕДЕРАЦИИ  
ПЕНЗЕНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

«УТВЕРЖДАЮ»

Зав. кафедрой МОиПЭВМ

А.Ю. Козлов

ЗАДАНИЕ

на курсовой проект по дисциплине

«Теория языков программирования и методы трансляции»

Тема: «Разработка транслятора»

ВАРИАНТ 4

На основании базового описания языка и в соответствии с вариантом задания разработать транслятор с заданного языка. Базовое описание языка имеет следующий вид:

<Программа> ::= <Объявление переменных> <Описание вычислений>  
<Описание вычислений> ::= BEGIN <список присваиваний> END  
<Объявление переменных> ::= VAR <список переменных> : тип ;  
<Список переменных> ::= <Идент>|<Идент>,<Список переменных>  
<Список присваиваний> ::= <Присваивание>|<Присваивание> <Список присваиваний>  
<Присваивание> ::= <Идент> = <Выражение>;  
<Выражение> ::= <Ун.оп.> <Подвыражение> | <Подвыражение>  
<Подвыражение> ::= (<Выражение>) | <Операнд> | <Подвыражение>  
<Бин.оп.> <Подвыражение>  
<Ун.оп> ::= вид  
<Бин.оп.> ::= вид  
<Операнд> ::= <Идент>|<Конст>  
<Идент> ::= <Буква><Идент>|<Буква>  
<Конст> ::= вид

Вариант задания представлен в таблицах 1.1 и 1.2.

Таблица 1.1 – Исходный вариант задания.

Тип переменных	Вид <Ун. оп.>	Вид <Бин. оп.>	Вид <конст.>	Максимальная длина идентификатора
LOGICAL	.NOT.	.AND.  .OR.  .EQU	0 1	11

Таблица 1.2 – Исходный вариант операторов

Операторы	Распознаватель
READ, IF ELSE, WRITE	Детерминированный восходящий распознаватель (магазинный автомат) для грамматики

Разработка должна проводиться на языке программирования C# в среде объектно-ориентированного программирования MS Visual Studio 2022.

Разработку выполнить с использованием операционной системы Windows 10.

Руководитель работы, к.т.н. доцент

Дорофеева О.С.

## Объём работы по курсу

### 1. Расчетная часть

- 1) Разработка структуры транслятора
  - 2) Проектирование лексического анализатора
  - 3) Проектирование магазинного автомата
  - 4) Разработка программных компонентов транслятора
- 
- 
- 

### 2. Графическая часть

- 1) Диаграмма деятельности
  - 2) Диаграмма классов
- 
- 
- 

### 3. Экспериментальная часть

Тестирование и отладка разработанного транслятора

---

---

---

## Срок выполнения проекта по разделам

1. Разработка структуры транслятора	к	23.09	2022 г.
2. Проектирование лексического анализатора	к	07.10	2022 г.
3. Проектирование магазинного автомата	к	21.10	2022 г.
4. Разработка программных компонентов	к	04.11	2022 г.
5. Тестирование и отладка	к	18.12	2022 г.
6. Оформление пояснительной записки	к	20.12	2022 г.

Дата выдачи задания " 7 " 09 2022 г.

Дата защиты проекта — 12 2022 г.

Руководитель Дорофеева О.С.

Задание получили " 7 " 09 2022 г.

Студент Левин М.В.

## Реферат

Пояснительная записка содержит 55 листов, 26 рисунков, 2 таблицы, 4 использованных источника, 2 приложения.

ВОСХОДЯЩИЙ РАЗБОР, ЯЗЫК ПРОГРАММИРОВАНИЯ, LR-АНАЛИЗАТОР, ТРАНСЛЯТОР, ИНТЕРПРЕТАТОР, ЛЕКСИЧЕСКИЙ АНАЛИЗ, СИНТАКСИЧЕСКИЙ АНАЛИЗ, МАГАЗИННЫЙ АВТОМАТ, ГРАММАТИЧЕСКИЙ РАЗБОР

Целью курсового проектирования является разработка учебного транслятора с заданного языка в форме интерпретатора.

Разработка проводилась на языке программирования C#.

Разработка проведена с использованием операционной системы Windows 10. Осуществлено функциональное тестирование разработанного транслятора, которое показало корректность его работы.

					ПГУ 09.03.04 – 05КР201.15ПЗ				
Изм.	Лис	№ докум.	Подп.	Дата	«Разработка транслятора»  Пояснительная записка	Лит.	Лист	Листов	
Разраб.		Левин М.В.							
Пров.		Дорофеева О.С.					4	55	
						Группа 20ВП1			
Н. контр.									
Утв.									

## Оглавление

Введение .....	2
1. Постановка задачи.....	3
2. Основные понятия и определения .....	5
2.1. Языки и грамматики .....	5
2.2. Структура транслятора.....	7
2.3. Синтаксический анализатор.....	9
3. Описание решения задачи .....	13
3.1. Преобразование грамматики.....	13
3.2. Лексический анализатор .....	14
3.3. LR(1) распознаватель .....	16
3.3.1. Построение управляющей таблицы.....	16
3.3.2. Управляющая программа .....	20
3.4. Семантический анализатор .....	22
4. Реализация программы .....	24
4.1. Кодирование .....	24
4.2. Диаграмма деятельности.....	25
4.3. Диаграмма классов .....	26
5. Функциональное тестирование .....	30
Заключение .....	32
Использованные источники.....	33
Приложение А .....	34
Приложение В .....	47

## Введение

По способу работы трансляторы делятся на компиляторы и интерпретаторы. В рамках данного курсового проекта будет реализован интерпретатор для языка, определенного соответствующей грамматикой.

Для разработки программы нужно решить следующие подзадачи:

1. Создание лексического анализатора. Его работа будет заключаться в извлечение списка лексем из исходного текста программы.
2. Создание синтаксического анализатора, представляющего собой детерминированный восходящий распознаватель, а именно LR(1)-распознаватель. Работая по принципу магазинного автомата, он будет формировать синтаксическое дерево.
3. Создание модуля, преобразующего синтаксическое дерево в абстрактное синтаксическое дерево (AST).
4. Создание модуля интерпретации, который, обходя AST, исполняет программу
5. Протестировать разработанную программу

Для реализации проекта будет использоваться Microsoft Visual Studio 2022 и язык программирования C#.

## 1. Постановка задачи

На основании базового описания языка и в соответствии с вариантом задания разработать транслятор с заданного языка. Базовое описание языка имеет следующий вид:

```

<Программа> ::= <Объявление переменных> <Описание вычислений>
<Описание вычислений> ::= BEGIN <список присваиваний> END
<Объявление переменных> ::= VAR <список переменных> : LOGICAL ;
<Список переменных> ::= <Идент>|<Идент>,<Список переменных>
<Список присваиваний> ::= <Присваивание>|<Присваивание> <Список
присваиваний>
<Присваивание> ::= <Идент> = <Выражение>;
<Выражение> ::= <Ун.оп.> <Подвыражение> | <Подвыражение>
<Подвыражение> ::= (<Выражение>) | <Операнд> | <Подвыражение>
<Бин.оп.><Подвыражение>
<Ун.оп> ::= NOT
<Бин.оп.> ::= AND | OR | EQU
<Операнд> ::= <Идент>|<Конст>
<Идент> ::= <Буква><Идент>|<Буква>
<Конст> ::= 0 | 1

```

Дополнить грамматику следующими операторами:

- READ (<Список переменных>);
- WRITE (<Список переменных>);
- IF (<Выражение>) THEN <Список присваиваний> ELSE <Список присваиваний> END\_IF

Дополнительные условия:

- <Буква> - буква латинского алфавита от A до Z;
- Максимальная длина идентификатора: 11

- Распознаватель: детерминированный восходящий



## 2. Основные понятия и определения

### 2.1. Языки и грамматики

**Цепочка символов  $\alpha$**  - произвольная упорядоченная конечная последовательность символов, записанных один за другим.

**Алфавит  $V$**  – это счётное множество допустимых символов языка.

**Языком  $L$**  над алфавитом  $V$  называется некоторое счётное подмножество цепочек конечной длины из множества всех цепочек над алфавитом  $V$ .

**Грамматика  $G$**  – это описание способа построения предложений некоторого языка.

**Правило** – это упорядоченная пара цепочек символов  $\alpha, \beta$ . В правилах важен порядок цепочек, поэтому их чаще записывают в вид  $\alpha \rightarrow \beta$  (или  $\alpha ::= \beta$ ). Такая запись читается как « $\alpha$  порождает  $\beta$ ». [1]

Грамматика  $G$  определяется как четвертка  $G(VT, VN, P, S)$ , где:

- **$VT$**  - множество терминальных символов, или алфавит терминальных символов
- **$VN$**  – множество нетерминальных символов, или алфавит нетерминальных символов;
- **$P$**  – множество правил грамматики вида  $\alpha \rightarrow \beta$
- **$S$**  – начальный символ грамматики,  $S \in VN$ .

**Классификация грамматик и языков [1]:**

1. **Грамматики обычного типа.** На структуру их правил не накладывается никаких ограничений:

2. **Контекстно-зависимые грамматики (КЗ-грамматики)** - грамматика, все правила которой имеют вид  $\alpha A \gamma \rightarrow \alpha \beta \gamma$  (у терминала имеется контекст, который сохраняется при его раскрытии)
3. **Контекстно-свободная грамматика (КС-грамматика)** — грамматика, все правила которой имеют вид  $A \rightarrow \beta$  (частный случай КЗГ, когда оба контекста пусты).
4. **Регулярные грамматики.** К типу регулярных относятся два эквивалентных класса грамматик: левостолбчатые и правостолбчатые. Правостолбчатые грамматики — грамматика, все правила которой имеют вид  $A \rightarrow aB$  или  $A \rightarrow \epsilon$  (справа либо пустая цепочка, либо терминал+нетерминал).

**Деревом вывода** грамматики  $G(VT, VN, P, S)$  называется дерево, которое соответствует некоторой цепочке вывода и удовлетворяет следующим условиям:

- Каждая вершина дерева обозначается символом грамматики  $A \in (VN \cup VT)$
- Корнем дерева является начальный символ  $S$
- Листьями дерева являются терминальные символы
- Если некоторый узел дерева обозначен нетерминальным символом  $A$ , а связанные с ним узлы – символами  $b_1, b_2, \dots, b_n$ , то в грамматике есть правило  $A \rightarrow b_1 b_2 \dots b_n$

### Пример [1]

$G: VT = \{ (, ) \} \quad VN = \{ S \}, \quad P = \{ S \rightarrow SS \mid (S) \mid \epsilon \}$

Слово:  $(( ))$

На рисунке 1 построено одно из возможных деревьев вывода

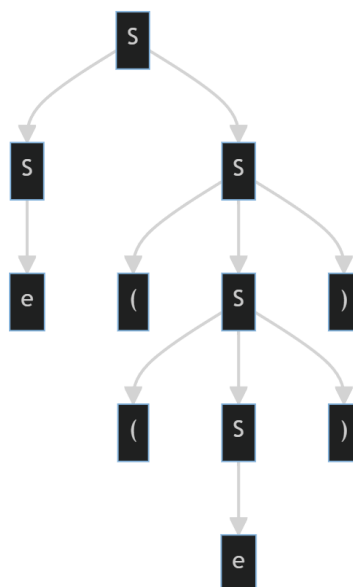


Рисунок 1 – Дерево вывода для цепочки  $()()$

## 2.2. Структура транслятора

**Транслятор** - это программа, которая считывает текст программы, написанной на одном языке (входном), и транслирует (переводит) его в эквивалентный текст на другом языке (выходном) [2].

Трансляция представляет собой три последовательных фазы анализа программы, на каждой из которой из текста программы извлекается все более «глубоко» скрытая информация о структуре и свойствах программы и ее объектах.

На рисунке 2 представлена условная схема работы транслятора [2]

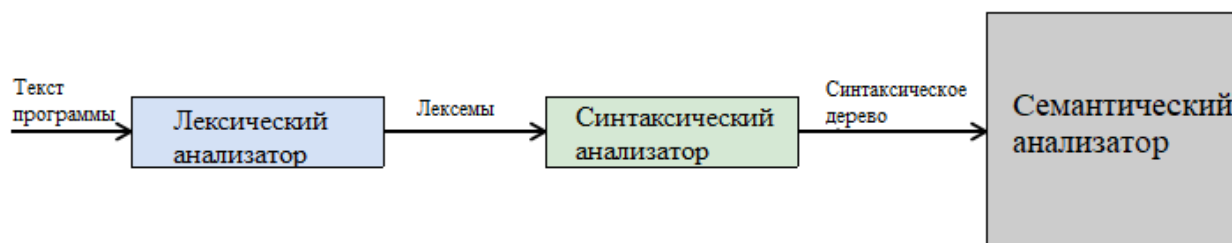


Рисунок 2 – Схема транслятора

Рассмотрим более подробно отдельные компоненты этого процесса, их свойства и взаимодействие между собой.

1. **Лексический анализ.** На этом этапе исходный текст программы «нарезается» на **лексемы** – последовательности входных символов (литер). Элементами лексики являются идентификаторы, константы, строки и т.п. Механизм распознавания базируется на простых системах распознавания, имеющих всего одну единицу памяти (текущее состояние) и табличное описание правил поведения (конечный автомат). Результат классификации лексем является входной информацией для синтаксического анализатора.
2. **Синтаксический анализ** - центральный элемент транслятора, который определяет принадлежность входной цепочки заданному языку. Синтаксис характеризует элементы, относящиеся к структуре программы: описания и определения данных, функций и других элементов программы. Операторы, выражения – это все синтаксические элементы разных уровней. Кроме того, вся программа представляет собой одно синтаксическое целое. Задачей синтаксического анализа является построение структуры – **синтаксического дерева**, отражающего взаимное расположение всех синтаксических элементов программы (их порядка приоритетов, вложенности). Средством описания синтаксиса являются **формальные грамматики**, а их свойства используются для построения распознавателей, базирующихся на табличном описании правил их поведения в сочетании со стековой памятью (**магазинные автоматы**).
3. **Семантический анализ** определяет смысловую нагрузку объектов анализируемой программы. Обходит синтаксическое дерево, полученное на предыдущей фазе, и выполняет соответствующие действия. Контекстно-свободные грамматики, с помощью которых описывают языки программирования, не позволяют задавать

контекстные условия, имеющиеся в любом языке. Следующие условия можно отнести к контекстным условиям:

- каждый используемый в программе идентификатор должен быть описан, но не более одного раза в одной зоне описания
- при вызове функции число фактических параметров и их типы должны соответствовать числу типов формальных параметров

## 2.3. Синтаксический анализатор

На основе **КС-грамматик** строятся синтаксические конструкции: описания типов и переменных, арифметические и логические выражения, управляющие операторы, и, наконец, полностью вся программа на входном языке [1].

Синтаксический анализатор – это часть компилятора, которая отвечает за выявление и проверку синтаксических конструкций входного языка. Синтаксический анализатор получает строку токенов от лексического анализатора, и проверяет, может ли эта строка токенов породиться грамматикой входного языка. Ещё одной функцией синтаксического анализатора является генерация сообщений об ошибках. В случае корректной программы синтаксический анализатор строит дерево разбора и передаёт его следующей части компилятора для дальнейшей обработки.

Основой для построения распознавателей КС-языков являются автоматы с **магазинной памятью** – МП-автоматы.

МП-автомат в отличие от обычного конечного автомата имеет стек, в который можно помещать специальные символы (обычно это терминальные и нетерминальные символы грамматики языка). Переход МП-автомата из одного состояния в другое зависит не только от входного символа, но и от одного или нескольких верхних символов стека. Таким образом,

конфигурация автомата определяется тремя параметрами: состоянием автомата, текущим символом входной цепочки и содержимым стека.

При выполнении перехода МП-автомата из одной конфигурации в другую из стека удаляются верхние символы, соответствующие условию перехода, и добавляется цепочка, соответствующая правилу перехода. Первый символ цепочки становится верхушкой стека.

В курсовом проекте будет реализовываться **детерминированный восходящий распознаватель**, а именно LR(k) распознаватель.

Восходящий анализатор предназначен для построения дерева разбора, начиная с листьев и двигаясь вверх к корню дерева разбора. Мы можем представить себе этот процесс как "свертку" исходной строки  $w$  к аксиоме грамматики. Каждый шаг свертки заключается в сопоставлении некоторой подстроки  $w$  и правой части какого-то правила грамматики и замене этой подстроки на нетерминал, являющийся левой частью правила. Если на каждом шаге подстрока выбирается правильно, то в результате мы получим правый вывод строки  $w$ .

**Пример.** Рассмотрим грамматику

$$S \rightarrow aABe$$

$$A \rightarrow Abc$$

$$A \rightarrow b$$

$$B \rightarrow d$$

Цепочка  $abbcde$  может быть свернута в аксиому следующим образом:

$$abbcde \rightarrow aAbcde \rightarrow aAde \rightarrow aABe \rightarrow S.$$

**LR(k)** означает, что [3]

- входная цепочка обрабатывается слева направо;
- выполняется правый вывод;

- не более  $k$  символов цепочки используются для принятия решения.

При LR(k)-анализе применяется метод "перенос-свертка" (shift-reduce). Этот метод использует магазинный автомат. Суть метода сводится к следующему. Символы входной цепочки переносятся в магазин до тех пор, пока на вершине магазина не накопится цепочка, совпадающая с правой частью какого-нибудь из правил (операция "перенос", "shift"). Далее все символы этой цепочки извлекаются из магазина и на их место помещается нетерминал, находящийся в левой части этого правила (операция "свертка", "reduce"). Входная цепочка допускается автоматом, если после переноса в автомат последнего символа входной цепочки и выполнения операции свертка, в магазине окажется только аксиома грамматик. [3]

Схематически структура LR(k)-анализатора изображена на рисунке 3.

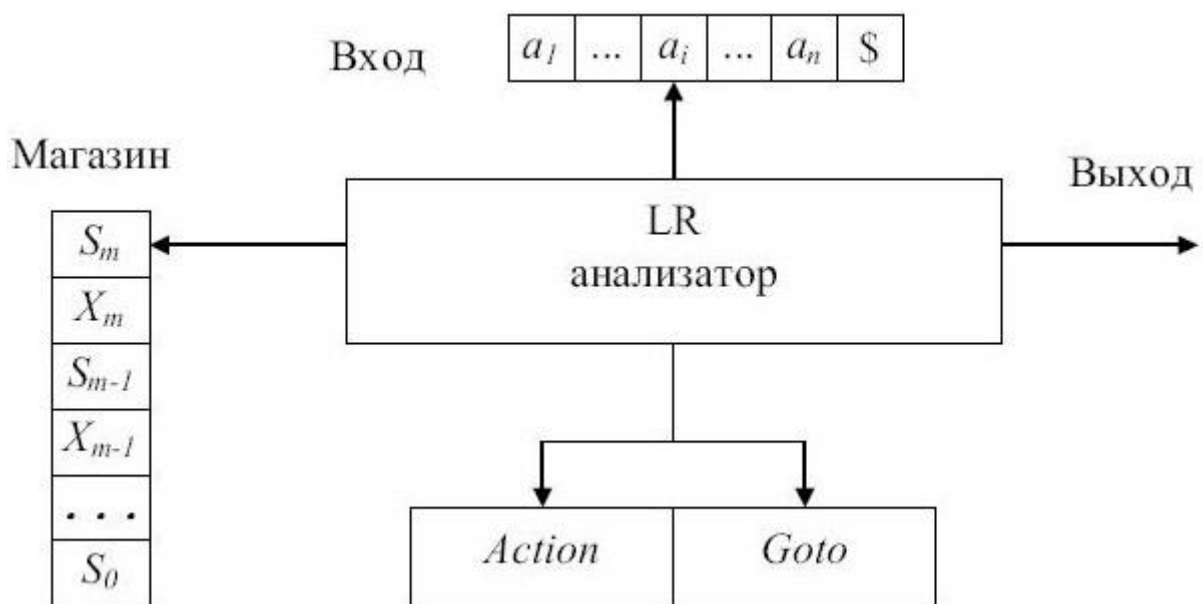


Рисунок 3 – структура LR(1) распознавателя

Анализатор состоит из входной ленты, выходной ленты, магазина, управляющей программы и таблицы, которая имеет две части - функцию действий ( Action ) и функцию переходов ( Goto ).

Элемент функции действий  $\text{Action}[s_m; a_i]$  для состояния  $s_m$  и входа  $a_i$  может иметь одно из четырех значений:

1. shift N (сдвиг) где N - символ состояния,
2. reduce  $A \rightarrow x$  (свертка по правилу грамматики  $A \rightarrow x$ )
3. accept (допуск),
4. error (ошибка)

Функция goto получает состояние и символ грамматики и выдает состояние. Функция goto, строящаяся по грамматике G, есть функция переходов детерминированного магазинного автомата, который распознает язык, порождаемый грамматикой G.

Элемент функции  $\text{Goto}[s_m; a_i]$  может иметь одно из двух значений:

1. S, где S - символ состояния,
2. error (ошибка).



### 3. Описание решения задачи

#### 3.1. Преобразование грамматики

Для удобства дальнейшей реализации заменим всю кириллицу, используемую в грамматике, на латиницу. Далее необходимо внести в нее изменения, дополнив следующими операторами: READ, WRITE, IF-ELSE. Для этого придется создать новый нетерминальный символ - **<operator>** и несколько правил вывода, где этот символ стоит в левой части:

- **<operator> ::= READ ( <list\_variables> ) ;**
- **<operator> ::= WRITE ( <list\_variables> ) ;**
- **<operator> ::= IF ( <expr> ) THEN <list\_assignments> ELSE <list\_assignments> END\_IF;**

Пока этот нетерминальный символ участвует только в левых частях правил грамматики, и не является начальным символом, а поэтому мы не сможем его достигнуть. Для успешного внесения нового нетерминала придется еще раз подправить грамматику. Изменим правило: **<description\_calculations> ::= BEGIN <list\_assignments> END** на **<description\_calculations> ::= BEGIN <list\_actions> END**. Заметим что появился новый нетерминальный символ - **<list\_actions>**, для которого были определены новые правила:

- **<list\_actions> ::= <list\_assignments> | <list\_assignments><list\_actions>**
- **<list\_actions> ::= <list\_operators> | <list\_operators><list\_actions>**

Дополнительно был добавлен новый нетерминальный символ **<list\_operators>**, для которого были определены новые правила:

- **<list\_operators> ::= <operator> | <operator><list\_operators>**

В конечном итоге, грамматика выглядит следующим образом.

**<program> ::= <variable\_declaration><description\_calculations>**

```

<description_calculations> ::= BEGIN <list_actions> END
<list_actions> ::= <list_assignments> | <list_assignments><list_actions>
<list_actions> ::= <list_operators> | <list_operators><list_actions>
<variable_declaration> ::= VAR <list_variables> : LOGICAL ;
<list_variables> ::= <id> | <id> , <list_variables>
<list_operators> ::= <operator> | <operator><list_operators>
<operator> ::= READ ( <list_variables> ) ;
<operator> ::= WRITE ( <list_variables> ) ;
<operator> ::= IF ( <expr> ) THEN <description_calculations> ELSE
<description_calculations> END_IF
<list_assignments> ::= <assignment> | <assignment><list_assignments>
<assignment> ::= <id> = <expr> ;
<expr> ::= <unary_op><sub_expr> | <sub_expr>
<sub_expr> ::= ( <expr> ) | <operand> | <sub_expr><bin_op><sub_expr>
<unary_op> ::= NOT
<bin_op> ::= AND | OR | EQU
<operand> ::= <id> | <const>
<id> ::= <letter><id>|<letter>
<const> ::= 0 | 1

```

### 3.2. Лексический анализатор

В основе лексических анализаторов лежат **регулярные грамматики**. Во многих языках программирования высокого уровня есть возможность работы с регулярным выражениями. **Регулярное выражение** - это шаблон, сопоставляемый с искомой строкой слева направо.

Для создания лексического анализатора нужно выделить класс лексем, которыми будет оперировать транслятор. **Лексема** – это структурная единица языка, которая состоит из элементарных символов языка и не содержит в своём составе других структурных единиц языка. Для выделения лексем

нужно им задать шаблоны (регулярное выражение), по которому лексер будет их вычленять.

Таблица 1 – Лексемы

Лексема	Шаблон
var	VAR
ident	\b[a-z]{1,11}\b
const	\b[01]\b
comma	,
colon	:
logical	LOGICAL
semicolon	;
begin	BEGIN
end	\bEND\b
space	[ \n\t\r]
assign	=
left_bracket	(
right_bracket	)
if	IF
end_if	END_IF
then	THEN
else	ELSE
read	READ
write	WRITE
not	NOT
and	AND
or	OR
equ	EQU
eof	\\$

### Пример

*BEGIN*

$x = y \text{ OR } (\text{NOT } 1);$

*END*

**Выделенные лексемы:** begin, ident, assign, ident, or, left\_bracket, not, const, right\_bracket, semicolon, end

## 3.3. LR(1) распознаватель

### 3.3.1. Построение управляющей таблицы

В начале нужно расширить грамматику, добавив новый начальный символ **S1** и новое правило **S1 → S**. Новая грамматика является эквивалентна предыдущей. Эти изменения вводятся для того, чтобы определить, когда анализатор должен остановить разбор и зафиксировать допуск входа. Таким образом, допуск имеет место тогда и только тогда, когда анализатор готов осуществить свертку по правилу **S1 → S**.

Множество префиксов правой сентенциальной формы, которые могут появиться в стеке при разборе методом «сдвиг-свертка» называются **активными префиксами**.

**LR(1) – ситуацией** называется пара  $[A \rightarrow \alpha \bullet \beta, a]$ , где  $A \rightarrow \alpha\beta$  – правило грамматики,  $a$  – терминал или правый концевой маркер \$. Вторая компонента ситуации называется аванцепочкой. [3]

LR(1) – ситуация  $[A \rightarrow \alpha \bullet \beta, a]$  допустима для активного префикса  $x$ , если существует вывод  $S \rightarrow^* \gamma A \omega \rightarrow \gamma \alpha \beta \omega$ , где  $x = \gamma \alpha$  и либо  $a$  – первый символ  $\omega$ , либо  $\omega = \varepsilon$  и  $a = \$$ .

Центральная идея метода заключается в том, что по грамматике строится детерминированный конечный автомат, распознающий активные префиксы.

Для этого ситуации группируются во множества, которые и образуют состояния автомата. Ситуации можно рассматривать как состояния недетерминированного конечного автомата, распознающего активные префиксы, а их группировка на самом деле есть процесс построения детерминированного конечного автомата из недетерминированного.

Анализатор, работающий слева-направо по типу сдвиг-свертка, должен уметь распознавать основы на верхушке магазина. Состояние автомата после прочтения содержимого магазина и текущий входной символ определяют очередное действие автомата. Функцией переходов этого конечного автомата является функция переходов LR-анализатора.

Чтобы не просматривать магазин на каждом шаге анализа, на верхушке магазина всегда хранится то состояние, в котором должен оказаться этот конечный автомат после того, как он прочитал символы грамматики в магазине от дна к верхушке. Рассмотрим ситуацию вида  $[A \rightarrow \alpha \bullet B \beta, a]$  из множества ситуаций, допустимых для некоторого активного префикса  $z$ . Тогда существует правосторонний вывод  $S \rightarrow^* \gamma A a \rightarrow \gamma \alpha B \beta a$ , где  $z = \gamma \alpha$ . Предположим, что из  $\beta a$  выводится терминальная строка  $bw$ . Тогда для некоторого правила вывода вида  $B \rightarrow q$  имеется вывод  $S \rightarrow^* z B b w \rightarrow z q b w$ . Таким образом ситуация  $[B \rightarrow \bullet q, b]$  также допустима для активного префикса  $z$  и ситуация  $[A \rightarrow \alpha B \bullet \beta, a]$  допустима для активного префикса  $zB$ . Терминальный символ  $b$  принадлежит множеству  $\text{FIRST}(\beta a)$ .

Система множеств допустимых LR(1)-ситуаций для всевозможных активных префиксов пополненной грамматики называется канонической системой множеств допустимых LR(1)-ситуаций. Для построения канонического набора нужно определить два новых множества: **closure(I)** и **goto(I, x)**.

Пусть  $I$  – множество пунктов грамматики  $G$ , тогда  $\text{closure}(I)$  представляет собой множество ситуаций, построенное из  $I$  согласно двум правилам.

1. Изначально в  $\text{closure}(I)$  добавляются все ситуации из  $I$ .
2. Если  $[A \rightarrow \alpha \bullet B \beta, a]$  входит в  $\text{closure}(I)$ , и существует правило  $B \rightarrow q$  то в  $\text{closure}(I)$  добавляется ситуация  $[B \rightarrow \bullet q, b]$  для каждого терминала  $b \in \text{FIRST}(\beta a)$ . Это правило применяется до тех пор, пока не останется пунктов, которые могут быть добавлены в  $\text{closure}(I)$ .

Множество переходов  $\text{goto}(I, X)$  (множество переходов из состояния  $I$  по символу  $X$ ) определяется как замыкание множества всех ситуаций  $[A \rightarrow \alpha X \bullet \beta, a]$ , таких что  $[A \rightarrow \alpha \bullet X \beta, a]$  находится в  $I$ .

Рассмотрим пример. Дана грамматики со следующими правилами:

$$S1 \rightarrow S$$

$$S \rightarrow CC$$

$$C \rightarrow cC \mid d$$

Вычисления замыкания  $\text{closure}(I_0)$  для начальной ситуации  $[S1 \rightarrow \bullet S, \$]$ . Заметим, что есть правило для  $S$ :  $S \rightarrow CC$ . Добавляем новую ситуацию  $[S \rightarrow \bullet CC, \$]$  в  $\text{closure}(I_0)$ .

Необходимо найти множество  $\text{FIRST}(C)$ . В это множество входят первые терминальные символы, выводимые из символа  $C$ . Следовательно  $\text{FIRST}(C) = \{c, d\}$ . Добавляем новые ситуации  $[C \rightarrow \bullet cC, c]$ ,  $[C \rightarrow \bullet cC, d]$ ,  $[C \rightarrow \bullet d, c]$ ,  $[C \rightarrow \bullet d, d]$  в  $\text{closure}(I_0)$ . Ни одна из новых ситуаций не имеет нетерминала непосредственно справа от точки, так что первое множество LR(1)-пунктов завершено. Аналогично строятся замыкания и для других пунктов.

На рисунке 4 показан LR(1) автомат для грамматики из примера выше. Стрелками обозначаются применение функции goto.

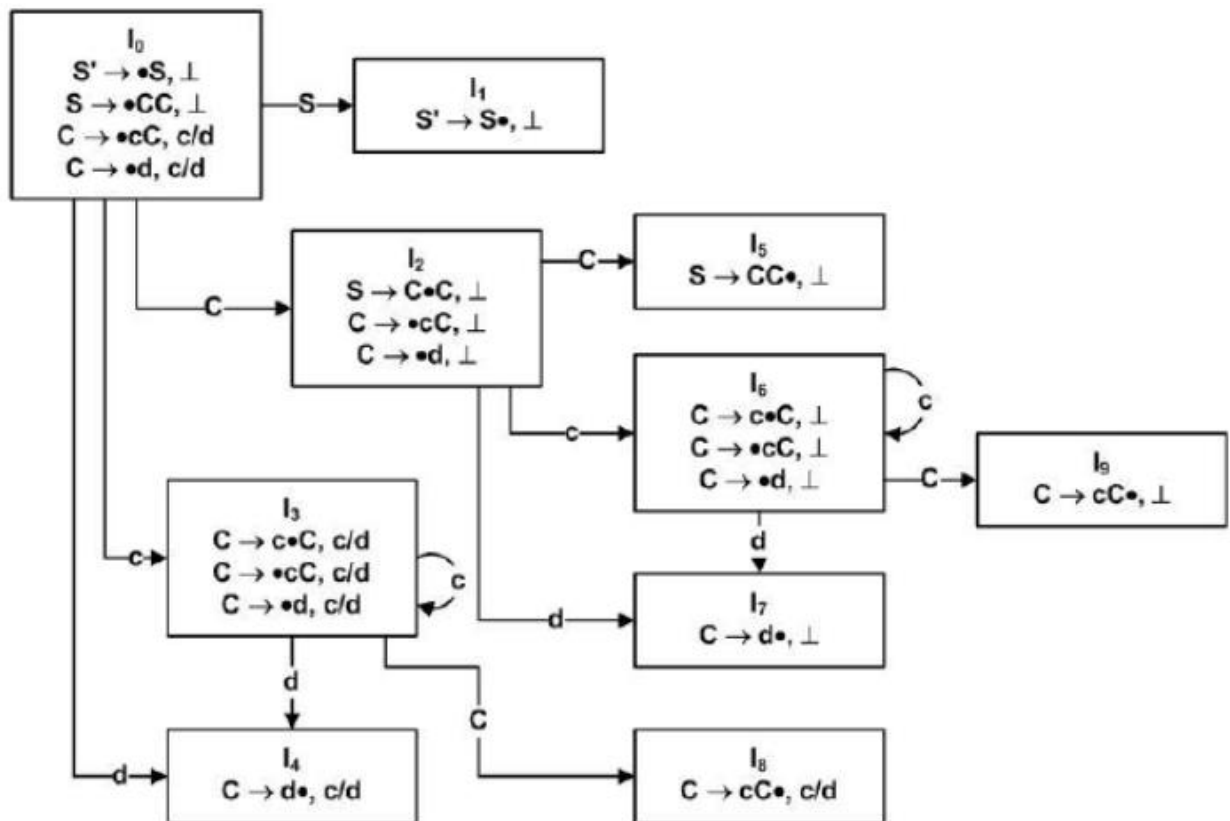


Рисунок 4 – LR(1) автомат

На основе полученного автомата строятся функции **ACTION** и **GOTO** следующим образом:

- Если  $[A \rightarrow \alpha \bullet a\beta, b]$  входит во множество ситуаций  $I_i$  и  $\text{goto}(I_i, a) = I_j$ , установить  $\text{ACTION}[i, a]$  равным shift j (сдвиг j).
- Если  $[A \rightarrow \alpha \bullet, a]$  входит во множество ситуаций  $I_i$  и  $A \neq S!$ , установить  $\text{ACTION}[i, a]$  равным reduce  $A \rightarrow \alpha$  (свертка по правилу).
- Если  $[S! \rightarrow S \bullet, \$]$  входит во множество ситуаций  $I_i$ , установить  $\text{ACTION}[i, \$]$  равным ассерт (принятие).

На основе этих функций строится таблица. Первый столбец представляет собой множества состояний (которые были получены с

помощью замыкания (closure)). Заголовки оставшихся столбцов представляют собой символы терминального и нетерминального словарей грамматики. В оставшихся ячейках хранятся значения функций **ACTION** и **GOTO**. Пример полученной таблицы продемонстрирован на рисунке 5.

Состояние	ACTION			GOTO	
	<i>c</i>	<i>d</i>	$\perp$	<i>S</i>	<i>C</i>
0	<i>s3</i>	<i>s4</i>		1	2
1			<i>acc</i>		
2	<i>s6</i>	<i>s7</i>			5
3	<i>s3</i>	<i>s4</i>			8
4	<i>r1</i>	<i>r3</i>			
5			<i>r1</i>		
6	<i>s6</i>	<i>s7</i>			9
7			<i>r3</i>		
8	<i>r2</i>	<i>r2</i>			
9			<i>r2</i>		

Рисунок 5 – Пример управляющей таблицы

### 3.3.2. Управляющая программа

Управляющая программа одинакова для всех LR-анализаторов, а таблица изменяется от одного анализатора к другому. Программа анализатора читает последовательно символы входной цепочки. Программа использует магазин для запоминания строки следующего вида  $s_0X_1s_1X_2...X_ms_m$ , где  $s_m$  - вершина магазина. Каждый  $X_i$  - символ грамматики,  $s_i$  - символ, называемый состоянием. Каждое состояние суммирует информацию, содержащуюся в стеке перед ним. Комбинация символа состояния на вершине магазина и текущего входного символа используется для индексирования управляющей таблицы и определения операции переноса-свертки. При реализации грамматические символы не обязательно располагаются в магазине. [3]

В начале работы управляющая программа получает таблицу с переходами и устанавливает считывающую головку на первый символ



входной цепочки. До тех пор, пока цепочка не будет обработана полностью, будут выполняться следующие действия:

- Если для входного символа **a** и текущего состояния **s** в таблице определено значение, равное **shift s1**, то в стек необходимо положить текущий символ и новое состояние **s1**. После чего необходимо перейти к следующему символу из входной цепочки
- Если в таблице для текущего символа и состояния определено значение, равное **reduce k**, то необходимо найти правило с номером **k** и извлечь из стека всю правую часть этого правила, а также сформировать очередной уровень синтаксического дерева. После чего положить в стек левую часть правила и продолжить работу без сдвига считывающей головки
- Если в таблице для текущего символа и состояния определено значение, равное **ассерт**, то это означает, что распознаватель успешно закончил свою работу и может вернуть сформировавшееся синтаксическое дерево.
- В противном случае, распознаватель завершается ошибкой.

на рисунке 6 показан узел синтаксического дерева для выражения **read(x, y, z, w)**

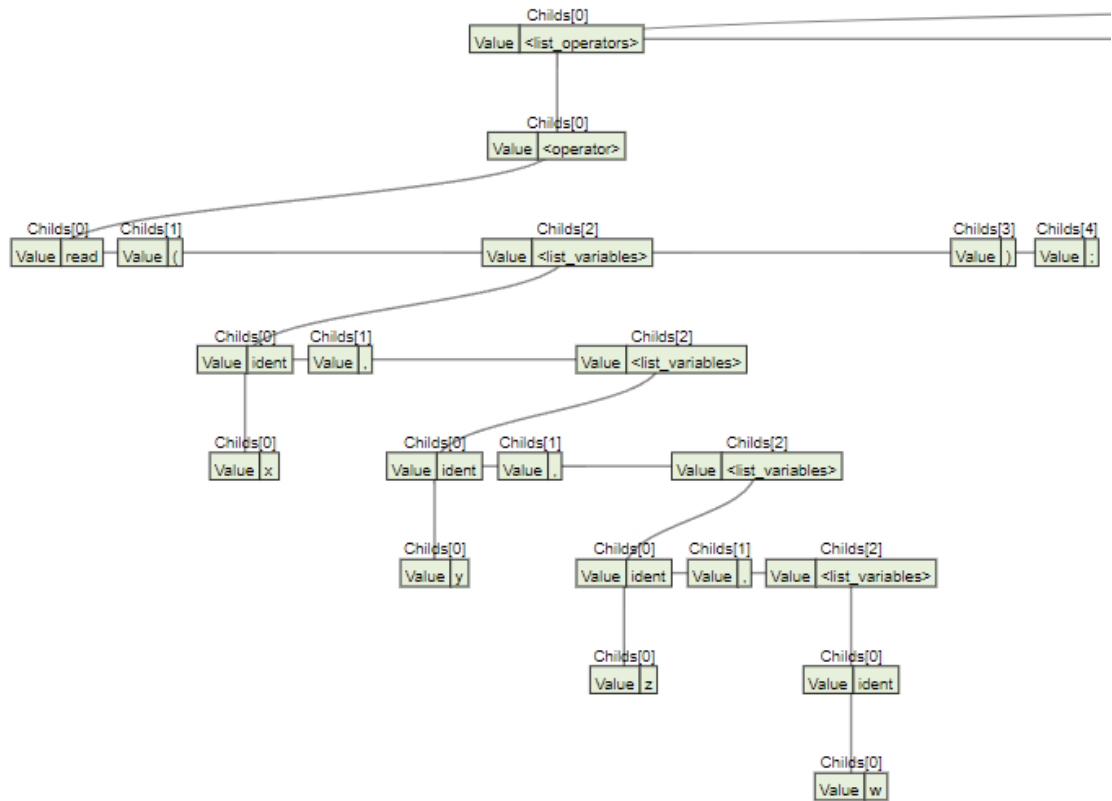


Рисунок 6 – Часть синтаксического дерева

### 3.4. Семантический анализатор

На этом этапе будет происходить непосредственное выполнение написанной программы. Для это необходимо полученное на предыдущем этапе дерево обойти сверху-вниз и слева направо.

Синтаксический анализатор строил дерево согласно правилам грамматики, из-за чего оно получилось с глубокой вложенностью, наличием ненужных лексем (например, «=», «+», «(», «)», «;» и т.д.) и отсутствием приоритета операций. Такое дерево будем называть **конкретным**. Для дальнейшей работы нам потребуется **абстрактное синтаксическое дерево (AST)** – дерево, которое содержит полную синтаксическую модель программы без лишних деталей.

Для построения абстрактного дерева нужно обойти конкретное и сформировать новые узлы, которые будут представлять классы,

соответствующие нетерминальным символам грамматики (например, для нетерминального символа <assignment> будет создан класс AssignmentNode и т.д. для остальных символов). Если рассматриваемый узел имеет значение, равное <assignment>, то нужно будет не забыть о наличии разного приоритета операций. На рисунке 7 показана часть AST для выражения **read(x, y, z, w)**.

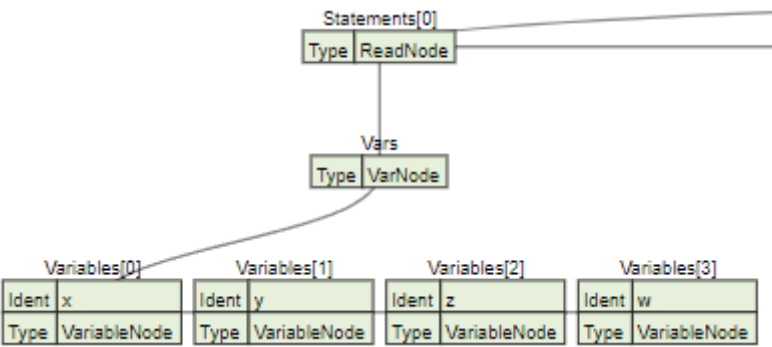


Рисунок 7 – Часть AST

## 4. Реализация программы

### 4.1. Кодирование

В результате выполнения поставленной задачи была разработана программа, исходный код которой приведен в Приложении А, а также он доступен по ссылке <https://github.com/martin1917/LRCompiler>

На рисунке 8 приведена структура готового проекта.

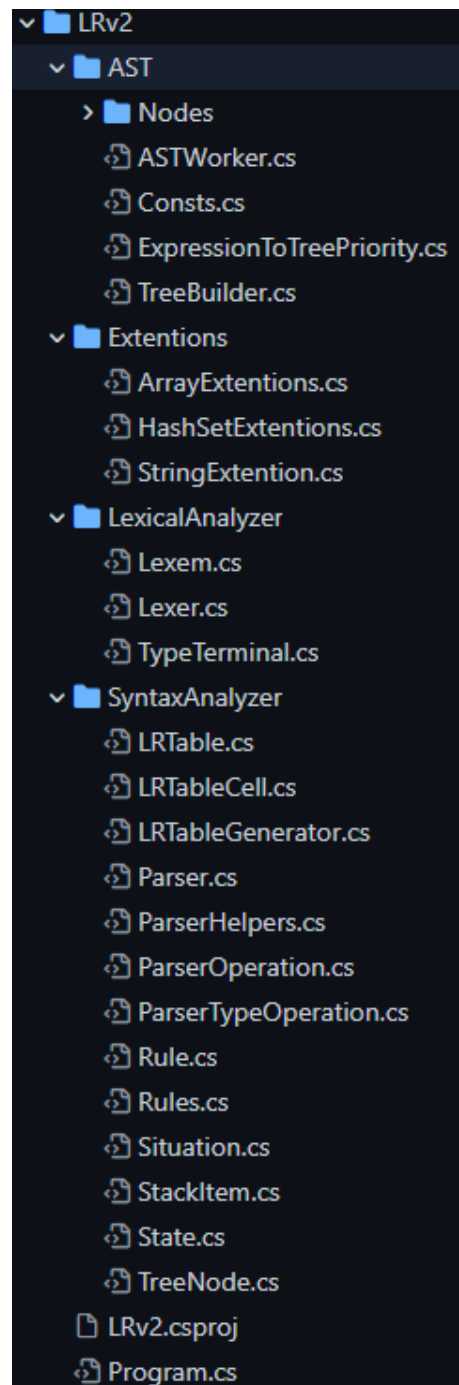


Рисунок 8 – структура проекта

## 4.2. Диаграмма деятельности

На рисунке 9 изображена диаграмма деятельности [4], отражающая логику работы восходящего распознавателя.

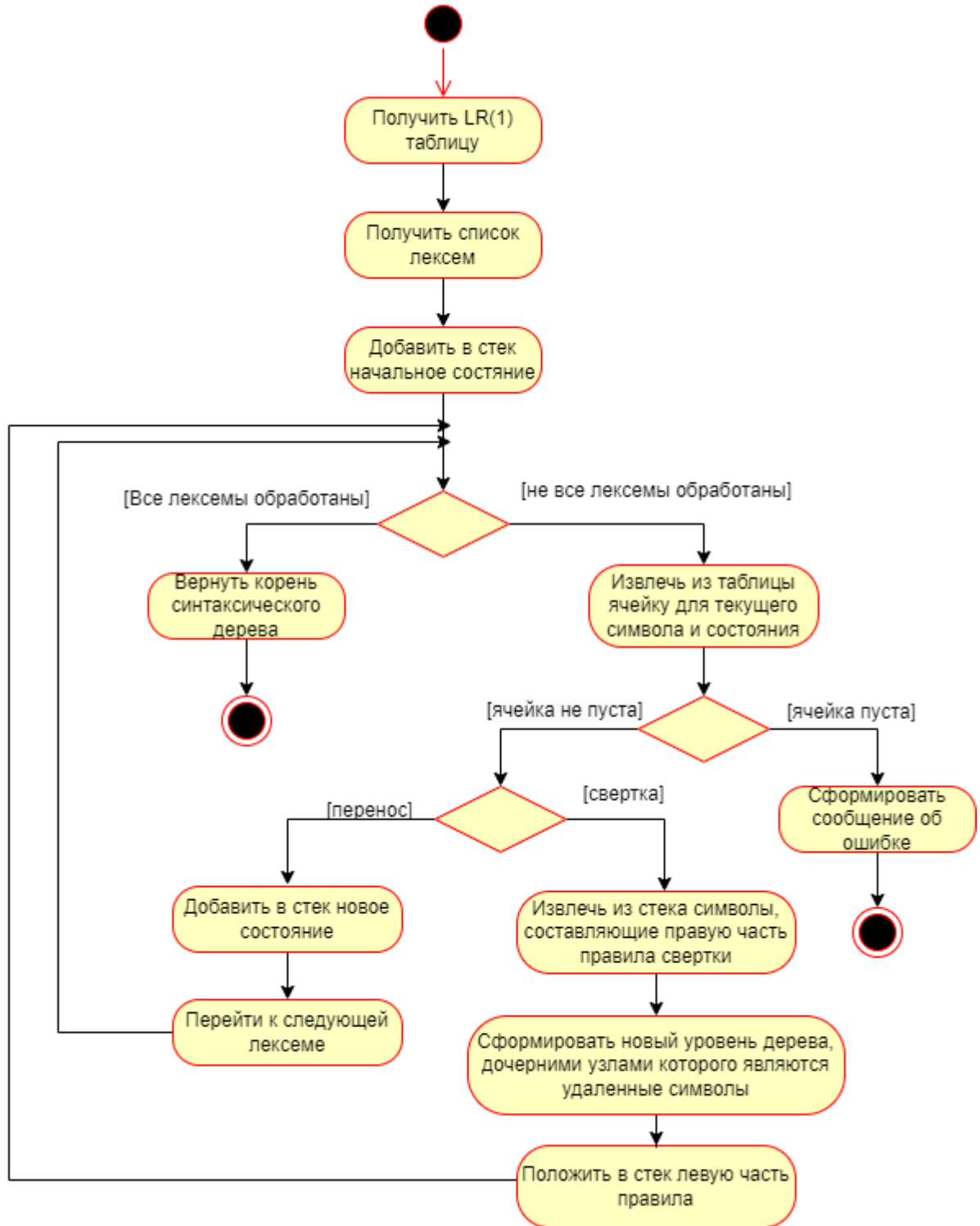


Рисунок 9 – диаграмма деятельности для восходящего распознавателя

### **4.3. Диаграмма классов**

На основе разработанного транслятора была сформирована диаграмма классов (рисунок 10). [4]



Краткое описание каждого класса:

- **TypeTerminal** – класс, определяющий типы лексем, а также регулярные выражения, распознающие их
- **Lexem** – класс, представляющий собой лексему. Содержит тип лексемы, значение лексемы, а также позицию, в которой она была найдена в тексте программы.
- **Lexer** – класс, представляющий собой лексический анализатор. Извлекает список лексем из переданного в него текста программы, или формирует ошибку, в случае не удачи.
- **ParserTypeOperation** – перечисление, содержащее типы возможных действий распознавателя (SHIFT - перенос, REDUCE - свертка, ACCPET - принятие, ERROR - ошибка).
- **ParserOperation** – структура, которая определяет тип операции распознавателя и хранит дополнительные данные, в виде целого числа. Для операции «перенос» это число определяет номер состояния магазинного автомата. Для операции «свертка» - номер правила, о котором будет происходить свертка.
- **LRTableCell** – класс, представляющий собой ячейку управляющей таблицы. Содержит номер состояния, лексему и соответствующую этим данным операцию распознавателя.
- **LRTable** – класс, представляющий собой управляющую таблицу.
- **LRTableGenerator** – класс, который конструирует таблицу для распознавателя.
- **Rule** – класс, представляющий собой правило грамматики.
- **Rules** – класс, представляющий собой коллекцию из всех правил грамматики.
- **Situation** – класс, представляющий собой LR(1) – ситуацию. Содержит правило грамматики, позицию точки, а также символ,



который должен идти после нетерминального символа, определяющего правило (левая часть правила).

- **State** – класс, представляющий собой состояние детерминированного восходящего распознавателя, которое получается в результате применения операции **замыкания**. Содержит список ситуаций.
- **TreeNode** – класс, представляющий собой узел синтаксического дерева. Содержит строку, представляющую символ грамматики, и дочерние узлы
- **Parser** – класс, представляющий собой распознаватель. Работает по принципу магазинного автомата. Содержит стек (магазин). По входному списку лексем и управляющей таблицы формирует синтаксическое дерево.
- **BaseNode** – класс, представляющий собой узел AST
- **TreeBuilder** – класс, который преобразует конкретное синтаксическое дерево в абстрактное
- **ExpressionToPriority** – класс, который строит узел для AST с учетом приоритета операций.
- **AstWorker** – класс, который непосредственно исполняет написанную программу. Обходит полученное на предыдущем шаге абстрактное синтаксическое дерево (AST) и выполняет команды, соответствующие узлам этого дерева.

## 5. Функциональное тестирование

В процессе разработки программы было проведено функциональное тестирование. Результаты тестирования представлены в таблице 2.

Таблица 2 – Результаты тестирования

Название теста	Параметры теста	Результат
Проверка выполнения корректной программы	Исходный код, соответствующий формальной грамматике	Тест пройден (Рис. В.1- В.2)
Проверка выполнения приоритета операция	Исходный код, соответствующий формальной грамматике.	Тест пройден (Рис. В.3- В.4)
Проверка корректности обработки недопустимых идентификаторов	Исходный код, у которого есть недопустимые идентификаторы	Тест пройден (Рис В.5- В.8)
Проверка корректности обработки необъявленных идентификаторов	Исходный код, в котором используются неизвестные идентификаторы	Тест пройден (Рис В.9- В.10)
Проверка корректности обработки нарушения скобочного баланса	Исходный код, в котором пропущена скобка	Тест пройден (Рис. В.11-В.12)
Проверка корректности обработки пропущенного ключевого слова	Исходный код, в котором пропущено ключевое слово END_IF	Тест пройден (Рис В.13- В.14)

Проверка корректности обработки ввода	Корректная программа, в которую передаются недопустимые значения для констант	Тест пройден (Рис В.15-В.16)
---------------------------------------	---	------------------------------

Тесты выше показывают, что транслятор работает корректно.

## Заключение

По итогу выполнения курсового проекта была написана программа, представляющая собой транслятор для языка, определенного соответствующей формальной грамматикой. Для этого были решены следующие подзадачи:

1. Разработан лексический анализатор
2. Разработан синтаксический анализатор
3. Разработан модуль перевода синтаксического дерева в абстрактное синтаксическое дерево
4. Разработан модуль интерпретации
5. Проведено функциональное тестирование

Были изучены способы преобразования грамматик и методы трансляции. Получены практические навыки в конструировании детерминированного восходящего распознавателя – LR(1) распознаватель.

## Использованные источники

1. Молдованова О.В. Языки программирования и методы трансляции: Учебное пособие. – Новосибирск/СибГУТИ, 2012 – 134с.
2. Касьянов В. Н., Поттосин И. В. Методы построения трансляторов. — Новосибирск: Наука, 1986. — 344 с.
3. Синтаксический анализ [Электронный ресурс]. URL: <https://intuit.ru/studies/courses/1157/173/lecture/4697> (дата обращения: 20.11.2022)
4. Джим Арлоу. UML2 и Унифицированный процесс. Практический объектно-ориентированный анализ и проектирование/Джим Арлоу, Айла Нейштадт. – Санкт-Петербург, Издательство Символ-Плюс, 2007. – 624с

## **Приложение А**

(листинг программы)

## LexicalAnalyzer/TypeTerminal.cs

```

using System.Reflection;
using System.Xml.Linq;

namespace LRV2.LexicalAnalyzer;
public class TypeTerminal
{
    private static List<TypeTerminal> allTypes;
    public static List<TypeTerminal> AllTypes => allTypes
        ??= LoadAllTypes();

    private static List<TypeTerminal> LoadAllTypes()
    {
        var fields = typeof(TypeTerminal)
            .GetFields(BindingFlags.Public | BindingFlags.Static)
            .Select(f => f.GetValue(null) as TypeTerminal);

        return new List<TypeTerminal>(fields!);
    }

    public static readonly TypeTerminal Var = new(name: "var", regex:
@"VAR");
    public static readonly TypeTerminal Ident = new(name: "ident", regex:
@"\b[a-z]{1,11}\b");
    public static readonly TypeTerminal Const = new(name: "const", regex:
@"\b[01]\b");
    public static readonly TypeTerminal Comma = new(name: ",", regex: @",");
    public static readonly TypeTerminal Colon = new(name: ":", regex: @":");
    public static readonly TypeTerminal Logical = new(name: "logical", regex:
@"LOGICAL");
    public static readonly TypeTerminal Semicolon = new(name: ";", regex: @";");
    public static readonly TypeTerminal Begin = new(name: "begin", regex:
@"BEGIN");
    public static readonly TypeTerminal End = new(name: "end", regex:
@"\bEND\b");
    public static readonly TypeTerminal Space = new(name: "space", regex: @"[
\n\t\r]");
    public static readonly TypeTerminal Assign = new(name: "=", regex: @"=");
    public static readonly TypeTerminal Lparam = new(name: "(", regex:
@"\(");
    public static readonly TypeTerminal Rparam = new(name: ")", regex:
@"\)");

```

```

    public static readonly TypeTerminal If          = new(name: "if",      regex:
@"IF");

    public static readonly TypeTerminal EndIf      = new(name: "end_if",  regex:
@"END_IF");

    public static readonly TypeTerminal Then       = new(name: "then",   regex:
@"THEN");

    public static readonly TypeTerminal Else       = new(name: "else",   regex:
@"ELSE");

    public static readonly TypeTerminal Read       = new(name: "read",   regex:
@"READ");

    public static readonly TypeTerminal Write      = new(name: "write",  regex:
@"WRITE");

    public static readonly TypeTerminal Not        = new(name: "not",    regex:
@"NOT");

    public static readonly TypeTerminal And        = new(name: "and",    regex:
@"AND");

    public static readonly TypeTerminal Or         = new(name: "or",     regex:
@"OR");

    public static readonly TypeTerminal Equ        = new(name: "equ",    regex:
@"EQU");

    public static readonly TypeTerminal Eof        = new(name: "eof",    regex:
@"\.$");

    public string Name { get; }

    public string Regex { get; }

    public override bool Equals(object? obj)
    {
        if (obj is not TypeTerminal typeTerminal) return false;
        return Name.Equals(typeTerminal.Name);
    }

    public override int GetHashCode()
    {
        return GetHashCode.Combine(Name);
    }

    public override string? ToString()
    {
        return Name;
    }

    private TypeTerminal(string name, string regex)

```



```

    {
        Name = name;
        Regex = regex;
    }
}

```

## LexicalAnalyzer/Lexem.cs

```

namespace LRV2.LexicalAnalyzer;
public class Lexem
{
    public TypeTerminal TypeTerminal { get; } = null!;
    public string Value { get; } = null!;
    public int Pos { get; }

    public Lexem(string value, TypeTerminal typeTerminal, int pos)
    {
        Value = value;
        TypeTerminal = typeTerminal;
        Pos = pos;
    }

    public string Type => TypeTerminal.Name;

    public bool IsIdentOrConst()
        => TypeTerminal.Name == TypeTerminal.Ident.Name
        || TypeTerminal.Name == TypeTerminal.Const.Name;

    public static Lexem CreateEndLexem()
        => new("$", TypeTerminal.Eof, -1);

    public override bool Equals(object? obj)
    {
        if (obj is not Lexem lexem) return false;
        return Value.Equals(lexem.Value) && TypeTerminal.Equals(lexem.TypeTerminal);
    }

    public override int GetHashCode()
    {
        return GetHashCode.Combine(Value, TypeTerminal);
    }
}

```

```

    }

    public override string? ToString()
    {
        return $"(type: {TypeTerminal}\\tvalue: {Value})";
    }
}

```

## LexicalAnalyzer/Lexer.cs

```

using System.Text.RegularExpressions;

namespace LRv2.LexicalAnalyzer;
public class Lexer
{
    private string text;
    private int pos;
    private List<Lexem> lexems = new();

    public Lexer(string text)
    {
        this.text = text;
    }

    public List<Lexem> GetLexems()
    {
        while (Next()) { }
        lexems.Add(Lexem.CreateEndLexem());
        return lexems;
    }

    public bool Next()
    {
        if (pos >= text.Length)
        {
            return false;
        }

        foreach (var type in TypeTerminal.AllTypes)
        {

```

```

        var regex = new Regex($"^{type.Regex}");
        var matches = regex.Matches(text[pos..]);
        if (matches.Count > 0)
        {
            var enumerator = matches.GetEnumerator();
            enumerator.MoveNext();
            var item = enumerator.Current.ToString();
            if (type != TypeTerminal.Space)
            {
                lexems.Add(new Lexem(item, type, pos));
            }
            pos += item.Length;
            return true;
        }
    }

    throw new Exception($"Не допустимая лексема на позиции {pos}");
}
}

```

## SyntaxAnalyzer/LRTableGenerator.cs

```

using LRv2.Extentions;
using LRv2.LexicalAnalyzer;

namespace LRv2.SyntaxAnalyzer;

public static class LRTableGenerator
{
    public static LRTable Generate()
    {
        LRTable table = new();

        var initSituation = new Situation(Rules.Init, 0, TypeTerminal.Eof.Name);
        var initState = new State(new List<Situation>() { initSituation });
        Closure(initState);

        var states = new List<State>() { initState };
        for (int i = 0; i < states.Count; i++)
        {

```

```

var state = states[i];
AddReducing(state, table);

var LexemsAfterDot = state.GetLexemsAfterDot();
foreach (var lexem in LexemsAfterDot)
{
    State newState = Goto(state, lexem);
    int pos = states.IndexOf(newState);
    int nextStateNumber;

    if (pos == -1)
    {
        states.Add(newState);
        nextStateNumber = newState.NumberState;
    }
    else
    {
        nextStateNumber = states[pos].NumberState;
    }

    var successAdding = table.Add(state.NumberState, lexem,
ParserOperation.NewShift(nextStateNumber));
    if (!successAdding)
    {
        var item = table.Cells.First(item => item.StateNumber ==
state.NumberState && item.InputingLexem == lexem);
        var message = $"[ERROR] - CONFLICT\n" +
            $"БЫЛО: {item}\n" +
            $"Пытаемся вставить 'SHIFT {nextStateNumber}'";

        throw new Exception(message);
    }
}

return table;
}

private static State Goto(State state, string lexem)
{

```

```

List<Situation> newSituations = new();
foreach (var situation in state.Situations)
{
    if (situation.DotPlaceAtEnd())
        continue;

    var lexemAfterDot = situation.GetLexemAfterDot();
    if (lexemAfterDot.Equals(lexem))
    {
        newSituations.Add(new Situation(situation.Rule, situation.Pos + 1,
situation.Lookahead));
    }
}

newSituations = newSituations.Distinct().ToList();
var newState = new State(newSituations);
Closure(newState);
return newState;
}

private static void Closure(State state)
{
    for (int i = 0; i < state.Situations.Count; i++)
    {
        var situation = state.Situations[i];

        if (situation.DotPlaceAtEnd())
            continue;

        var lexem = situation.GetLexemAfterDot();

        if (!lexem.IsNotTerminal())
            continue;

        var rules = Rules.GetStartWith(lexem);
        foreach (var rule in rules)
        {
            if (situation.AfterDotOnlyOneLexem())
            {
                state.AddSituation(new Situation(rule, 0, situation.Lookahead));
            }
        }
    }
}

```

```

        }
        else
        {
            var firstLexems =
ParserHelpers.FirstLexemsFor(situation.Rule.Right[situation.Pos + 1]);
            foreach (var first in firstLexems)
            {
                state.AddSituation(new Situation(rule, 0, first));
            }
        }
    }
}

private static void AddReducing(State state, LRTable table)
{
    foreach (var situation in state.Situations)
    {
        if (!situation.DotPlaceAtEnd())
            continue;

        if (situation.Rule.Left.StartsWith("<S>"))
        {
            table.Add(state.NumberState, "eof", ParserOperation.NewAccept());
        }
        else
        {
            var successAdding = table.Add(state.NumberState, situation.Lookahead,
ParserOperation.NewReduce(situation.Rule.NumberRule));
            if (!successAdding)
            {
                var item = table.Cells.First(item => item.StateNumber ==
state.NumberState && item.InputingLexem == situation.Lookahead);
                var message = $"ERROR - CONFLICT\n" +
                    $"БЫЛО: {item}\n" +
                    $"Пытаемся вставить 'REDUCE {situation.Rule.NumberRule}'";
                throw new Exception(message);
            }
        }
    }
}
}

```

```
}
```

## SyntaxAnalyzer/ParserHelpers.cs

```
using LRv2.Extentions;

namespace LRv2.SyntaxAnalyzer;
public static class ParserHelpers
{
    public static HashSet<string> FirstLexemsFor(string lexem)
    {
        HashSet<string> result = new();

        if (!lexem.IsNotTerminal())
        {
            result.Add(lexem);
            return result;
        }

        var rules = Rules.GetStartWith(lexem);

        foreach (var rule in rules)
        {
            var rightFirst = rule.Right[0];

            if (rightFirst.Equals(lexem)) continue;

            result.AddRange(FirstLexemsFor(rightFirst));
        }

        return result;
    }

    public static HashSet<string> FollowLexemsFor(string lexem)
    {
        var rules = Rules.GetRightPartContains(lexem);
```

```

HashSet<string> result = new();

foreach (var rule in rules)
{
    var indexes = rule.Right.IndexesOf(lexem);
    foreach (var index in indexes)
    {
        if (index == rule.Right.Length - 1)
        {
            if (rule.Left.StartsWith(rule.Right[index])) continue;
            result.AddRange(FollowLexemsFor(rule.Left));
        }
        else
        {
            result.AddRange(FirstLexemsFor(rule.Right[index + 1]));
        }
    }
}

return result;
}
}

```

## SyntaxAnalyzer/Parser.cs

```

using LRv2.LexicalAnalyzer;
using System.Data;

namespace LRv2.SyntaxAnalyzer;

public class Parser
{
    private readonly LRTable table;

    public Parser(LRTable table)
    {
        this.table = table;
    }

    public TreeNode Parse(List<Lexem> lexems, bool needLogging = false)

```



```

{
    bool accept = false;
    int i = 0;

    var stack = new Stack<StackItem>();
    stack.Push(new StackItem(0, ""));

    while (!accept)
    {
        var stateOnTopStack = stack.Peek();
        var parserOperation = table.Get(stateOnTopStack.StateNumber,
lexems[i].Type);

        if (parserOperation.TypeOperation is ParserTypeOperation.ERROR)
        {
            var follow = ParserHelpers.FollowLexemsFor(stateOnTopStack.Value);

            var message =
                $"После '{stateOnTopStack.Value}' должны быть следующие символы
[{string.Join(", ", follow)}]\n" +
                $"А никак НЕ '{lexems[i].TypeTerminal.Name}';

            throw new Exception(message);
        }

        switch (parserOperation.TypeOperation)
        {
            case ParserTypeOperation.ACCEPT:
            {
                accept = true;
            }
            break;

            case ParserTypeOperation.SHIFT:
            {
                var nextStateNumber = parserOperation.Number;

                StackItem stackItem = lexems[i].IsIdentOrConst()
                    ? new StackItem(nextStateNumber, lexems[i].Type,
lexems[i].Value)
                    : new StackItem(nextStateNumber, lexems[i].Type);

```

```

        stack.Push(stackItem);

        i++;
    }

    break;

case ParserTypeOperation.REDUCE:
{
    var rule = Rules.GetByNumber(parserOperation.Number);

    var childs = new List<TreeNode>();

    for (int k = 0; k < rule.Right.Length; k++)
    {
        StackItem stackItem = stack.Pop();

        TreeNode? child = stackItem.TreeNode;
        child ??= stackItem.Payload == null
            ? new TreeNode(stackItem.Value)
            : new TreeNode(stackItem.Value, new() { new
TreeNode(stackItem.Payload) });

        childs.Insert(0, child);
    }

    var stateAferReducing = stack.Peek().StateNumber;
    var operation = table.Get(stateAferReducing, rule.Left);
    var nextStateNumber = operation.Number;

    stack.Push(new StackItem(nextStateNumber, rule.Left, new
TreeNode(rule.Left, childs)));
    }

    break;
}

return stack.Pop().TreeNode!;
}
}

```

## **Приложение В**

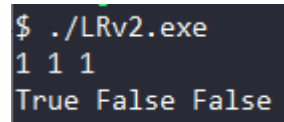
(тестирование)

```

1  VAR
2      x, y, z: LOGICAL;
3
4  BEGIN
5      READ(x, y, z);
6      x = y OR z;
7      y = NOT(x EQU x AND y);
8
9      IF (z)
10     THEN
11         BEGIN
12             z = 0;
13         END
14     ELSE
15         BEGIN
16             z = 1;
17         END
18     END_IF
19
20     WRITE (x, y, z);
21 END

```

Рисунок В.1 – Корректная программа 1



```

$ ./LRv2.exe
1 1 1
True False False

```

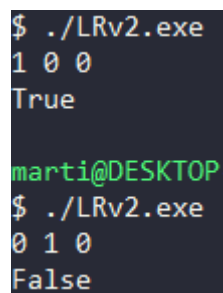
Рисунок В.2 – Результат выполнения корректной программы 1

```

1  VAR
2      x, y, z: LOGICAL;
3
4  BEGIN
5      READ(x, y, z);
6      z = x OR y AND z;
7      WRITE(z);
8  END

```

Рисунок В.3 – Корректная программа 2



```

$ ./LRv2.exe
1 0 0
True

marti@DESKTOP
$ ./LRv2.exe
0 1 0
False

```

Рисунок В.4 – Влияние приоритета операции на ответ

```

1  VAR
2    x, y, z14: LOGICAL;
3
4  BEGIN
5    READ(x, y, z);
6    x = y OR z;
7    y = NOT(x EQU x AND y);
8
9    IF (z)
10   THEN
11     BEGIN
12       z = 0;
13     END
14   ELSE
15     BEGIN
16       z = 1;
17     END
18   END_IF
19
20   WRITE (x, y, z);
21 END

```

Рисунок В.5 – Недопустимые символы в идентификаторе

```

$ ./LRv2.exe
Не допустимая лексема на позиции 12

```

Рисунок В.6 – Корректная обработка идентификатора с недопустимыми символами

```

1  VAR
2    xqwewqweewqeqeewweweqweqweq: LOGICAL;
3
4  BEGIN
5    READ(x, y, z);
6    x = y OR z;
7    y = NOT(x EQU x AND y);
8
9    IF (z)
10   THEN
11     BEGIN
12       z = 0;
13     END
14   ELSE
15     BEGIN
16       z = 1;
17     END
18   END_IF
19
20   WRITE (x, y, z);
21 END

```

Рисунок В.7 – Превышение допустимой длины идентификатора

```

$ ./LRv2.exe
Не допустимая лексема на позиции 6

```

Рисунок В.8 – Корректная обработка слишком длинного идентификатора

```

1  VAR
2    x, y: LOGICAL;
3
4  BEGIN
5    READ(x, y, z);
6    x = y OR z;
7    y = NOT(x EQU x AND y);
8
9    IF (z)
10   THEN
11     BEGIN
12       z = 0;
13     END
14   ELSE
15     BEGIN
16       z = 1;
17     END
18   END_IF
19
20   WRITE (x, y, z);
21 END

```

Рисунок В.9 – Использование не объявленного идентификатора z

```

$ ./LRv2.exe
Переменная z не объявлена
Переменная z не объявлена
Переменная z не объявлена
Переменная z не объявлена
Переменная z не объявлена
Переменная z не объявлена

```

Рисунок В.10 – Корректная обработка неизвестного идентификатора

```

1  VAR
2    x, y, z: LOGICAL;
3
4  BEGIN
5    READ(x, y, z);
6    x = y OR z;
7    y = NOT x EQU x AND y);
8
9    IF (z)
10   THEN
11     BEGIN
12       z = 0;
13     END
14   ELSE
15     BEGIN
16       z = 1;
17     END
18   END_IF
19
20   WRITE (x, y, z);
21 END
22

```

Рисунок В.11 – Пропущена открывающая скобка

```

$ ./LRv2.exe
После 'ident' должны быть следующие символы [:, ), ,, =, ;, and, or, equ]
А никак НЕ ')'

```

Рисунок В.12 – Корректная обработка пропущенной скобки

```

1  VAR
2      x, y, z: LOGICAL;
3
4  BEGIN
5      READ(x, y, z);
6      x = y OR z;
7      y = NOT(x EQU x AND y);
8
9      IF (z)
10     THEN
11         BEGIN
12             z = 0;
13         END
14     ELSE
15         BEGIN
16             z = 1;
17         END
18
19
20     WRITE (x, y, z);
21 END

```

Рисунок В.13 – Пропущено ключевое слово END\_IF

```

$ ./LRv2.exe
После 'end' должны быть следующие символы [else, end_if]
А никак НЕ 'write'

```

Рисунок В.14 – Корректная обработка пропущенного ключевого слова

```

1  VAR
2      x, y, z: LOGICAL;
3
4  BEGIN
5      READ(x, y, z);
6      z = x OR y AND z;
7      WRITE(z);
8  END

```

Рисунок В.15 – Корректная программа

```

$ ./LRv2.exe
1 15
- Функции READ нужно передать значения для переменных x, y, z
- Значением переменных могут быть только 0 или 1

```

Рисунок В.16 – Ввод некорректных данных в корректную программу