# Содержание

%TOC%

# Введение

Язык программирования — это формальный язык, предназначенный для записи компьютерных программ. Однозначно определить такой язык можно с помощью соответствующей спецификации, однако сам по себе он не подлежит использованию, ведь компьютер не умеет выполнять текст. Для решения задачи перевода такого текста в понятный компьютеру вид необходим компилятор — программа, осуществляющая трансляцию исходного текста, написанного в соответствии с грамматикой целевого языка программирования в соответствующую последовательность команд.

В современном мире довольно удобно работать с компиляторами, состоящими из двух частей, называемых препроцессором и постпроцессором.

В Препроцессор включают следующие этапы разбора программы:

* лексический анализ;
* синтаксический анализ;
* семантический анализ.

Результатом его работы является дерево, представляющее синтаксическую структуру исходного текста. Это дерево хранится в основной памяти и образует интерфейс для постпроцессора, который может реализовывать как генерацию кода для конкретной архитектуры процессора или виртуальной машины, так и трансляцию на другой язык программирования.

Преимущество данного разделения в возможности сочетания различных препроцессоров с разными постпроцессорами, в зависимости от задачи.

В данной курсовой работе исходными данными является уникальная грамматика модельного языка программирования и актуальность работы обуславливается отсутствием существующих распознавателей текстов, соответствующих данной грамматике.

В связи с вышеописанной проблемой возникает необходимости в создании такого продукта, который бы реализовывал такой разбор.

В продукте будет реализован один главный препроцессор и 2 разных постпроцессора, реализующих трансляцию на такие языки программирования как Python и C++.

Ожидаемый результат работы продукта — функционирующий транслятор программы на модельном языке программирования в программы на таких языках, как Python и C++ с информативными сообщениями о структурных ошибках в исходном тексте.

Цель курсовой работы состоит в:

* закреплении теоретических знаний в области теории формальных языков, грамматик и автоматов;
* формирование практических умений и навыков разработки собственного распознавателя модельного языка программирования;
* закрепление практических навыков самостоятельного решения инженерных задач, развитие творческих способностей студентов и умений пользоваться технической, нормативной и справочной литературой.

Для разработки данного продукта требуется решить следующие основные задачи:

Какие тут стоит писать задачи, те, что я написал ниже? Я продублировал их в следующем разделе.

1. В соответствии с номером варианта составить формальное описание модельного языка программирования с помощью:

* РБНФ;
* диаграмм Вирта;
* формальных грамматик.

1. Разработать спецификацию языка программирования.
2. Написать пять содержательных примеров программ, раскрывающих особенности конструкций модельного языка программирования, отразив в этих примерах все его функциональные возможности.
3. Составить таблицы лексем и диаграмму состояний с действиями для распознавания и формирования лексем языка.
4. По диаграмме с действиями написать функцию сканирования текста входной программы на модельном языке.
5. Разработать программное средство, реализующее лексический анализ текста программы на входном языке.
6. Реализовать синтаксический анализатор текста программы на модельном языке методом рекурсивного спуска.
7. Построить цепочку вывода и дерево разбора простейшей программы на модельном языке из начального символа грамматики.
8. Дополнить синтаксический анализатор процедурами проверки семантической правильности программы на модельном языке в соответствии с контекстными условиями вашего варианта.
9. Распечатать пример таблиц идентификаторов и двуместных операций.
10. Показать динамику изменения содержимого стека при семантическом анализе программы на примере одного синтаксически правильного выражения.
11. Синтаксический анализатор процедурами построения синтаксического дерева.
12. Реализовать постпроцессор, преобразующий синтаксическое дерево в программу на языке Python.
13. Реализовать постпроцессор, преобразующий синтаксическое дерево в программу на языке C++.
14. Составить набор контрольных примеров, демонстрирующих все возможные типы лексических, синтаксических и семантических ошибок в программах на модельном языке.
15. Составить набор примеров трансляции программ на модельном языке программирования на языки Python и C++.

# Постановка задачи

Для разработки транслятора языка программирования требуется решить следующие основные задачи:

1. В соответствии с номером варианта составить формальное описание модельного языка программирования с помощью:

* РБНФ;
* диаграмм Вирта;
* формальных грамматик.

1. Разработать спецификацию языка программирования.
2. Написать пять содержательных примеров программ, раскрывающих особенности конструкций модельного языка программирования, отразив в этих примерах все его функциональные возможности.
3. Составить таблицы лексем и диаграмму состояний с действиями для распознавания и формирования лексем языка.
4. По диаграмме с действиями написать функцию сканирования текста входной программы на модельном языке.
5. Разработать программное средство, реализующее лексический анализ текста программы на входном языке.
6. Реализовать синтаксический анализатор текста программы на модельном языке методом рекурсивного спуска.
7. Построить цепочку вывода и дерево разбора простейшей программы на модельном языке из начального символа грамматики.
8. Дополнить синтаксический анализатор процедурами проверки семантической правильности программы на модельном языке в соответствии с контекстными условиями вашего варианта.
9. Распечатать пример таблиц идентификаторов и двуместных операций.
10. Показать динамику изменения содержимого стека при семантическом анализе программы на примере одного синтаксически правильного выражения.
11. Синтаксический анализатор процедурами построения синтаксического дерева.
12. Реализовать постпроцессор, преобразующий синтаксическое дерево в программу на языке Python.
13. Реализовать постпроцессор, преобразующий синтаксическое дерево в программу на языке C++.
14. Составить набор контрольных примеров, демонстрирующих все возможные типы лексических, синтаксических и семантических ошибок в программах на модельном языке.
15. Составить набор примеров трансляции программ на модельном языке программирования на языки Python и C++.

# Формальная модель задачи

Существуют три основных метода описания синтаксиса языков программирования: формальные грамматики, формы Бэкуса-Наура и диаграммы Вирта.

Описание грамматики с помощью РБНФ будет иметь вид, изображённый листинге [3.1](#lst:1.1).

Листинг 3.1 – Грамматика, записанная в РБНФ

<программа>::= '{' {/ (<описание> | <оператор>) ';' /} '}'  
  
<описание>::= <тип> <идентификатор> { ',' <идентификатор> }  
  
<тип>::= 'int' | 'float' | 'bool'  
  
<оператор>::= <составной> |  
 <присваивания> |  
 <условный> |  
 <фиксированного\_цикла> |  
 <условного\_цикла> |  
 <ввода> |  
 <вывода>  
  
<составной>::= 'begin' <оператор> { ';' <оператор> } 'end'  
  
<присваивания>::= <идентификатор> ':=' <выражение>  
  
<условный>::= 'if' '(' <выражение> ')' <оператор> ['else' <оператор>]  
  
<фиксированного\_цикла>::= 'for' <присваивания> 'to' <выражение> ['step' <выражение>] <оператор> 'next'  
  
<условного\_цикла>::= 'while' '(' <выражение> ')' <оператор>  
  
<ввода>::= 'readln' <идентификатор> { ',' <идентификатор> }  
  
<вывода>::= 'writeln' <выражение> { ',' <выражение> }  
  
  
<выражение> ::= <сумма> { <операции\_группы\_отношения> <сумма>}  
  
<сумма> ::= <произведение> { <операции\_группы\_сложения> <произведение>}  
  
<произведение> ::= <множитель> { <операции\_группы\_умножения> <множитель>}  
  
<множитель> ::= <идентификатор> |  
 <число> |  
 <логическая\_константа> |  
 <унарная\_операция> <множитель> |  
 '(' <выражение> ')'  
  
  
<логическая\_константа>:: = 'true' |  
 'false'  
  
<операции\_группы\_отношения>::= '!=' |  
 '==' |  
 '<' |  
 '<=' |  
 '>' |  
 '>='  
  
<операции\_группы\_сложения>::= '+' |  
 '-' |  
 '||'  
  
<операции\_группы\_умножения>::= '\*' |  
 '/' |  
 '&&'  
  
<унарная\_операция>::= '!'  
  
<идентификатор> ::= <буква> { <буква> | <цифра> }  
  
<число>::= <целое> |  
 <действительное>  
  
<целое>::= <двоичное> |  
 <восьмеричное> |  
 <десятичное> |  
 <шестнадцатеричное>  
  
<двоичное>::= {/ '0' | '1' /} ('B' | 'b')  
  
<восьмеричное>::= {/ '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' /} ('O' | 'o')  
  
<десятичное>::= {/ <цифра> /} ['D' | 'd']  
  
<шестнадцатеричное>::= <цифра> { <цифра> | 'A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'a' | 'b' | 'c' | 'd' | 'e' | 'f' } ('H' | 'h')  
  
<действительное>::= <числовая\_строка> <порядок> |  
 [<числовая\_строка>] '.' <числовая\_строка> [порядок]  
  
<числовая\_строка>::= {/ <цифра> /}  
  
<порядок>::= ('E' | 'e')['+' | '-'] <числовая\_строка>  
  
<буква> :: = 'a' |  
 'b' |  
 'c' |  
 'd' |  
 'e' |  
 'f' |  
 'g' |  
 'h' |  
 'i' |  
 'j' |  
 'k' |  
 'l' |  
 'm' |  
 'n' |  
 'o' |  
 'p' |  
 'q' |  
 'r' |  
 's' |  
 't' |  
 'u' |  
 'v' |  
 'w' |  
 'x' |  
 'y' |  
 'z' |  
 'A' |  
 'B' |  
 'C' |  
 'D' |  
 'E' |  
 'F' |  
 'G' |  
 'H' |  
 'I' |  
 'J' |  
 'K' |  
 'L' |  
 'M' |  
 'N' |  
 'O' |  
 'P' |  
 'Q' |  
 'R' |  
 'S' |  
 'T' |  
 'U' |  
 'V' |  
 'W' |  
 'X' |  
 'Y' |  
 'Z'  
  
<цифра> ::= '0' |  
 '1' |  
 '2' |  
 '3' |  
 '4' |  
 '5' |  
 '6' |  
 '7' |  
 '8' |  
 '9'

Грамматика в виде диаграммы Вирта приведена на рисунке [3.1](#fig:virt)

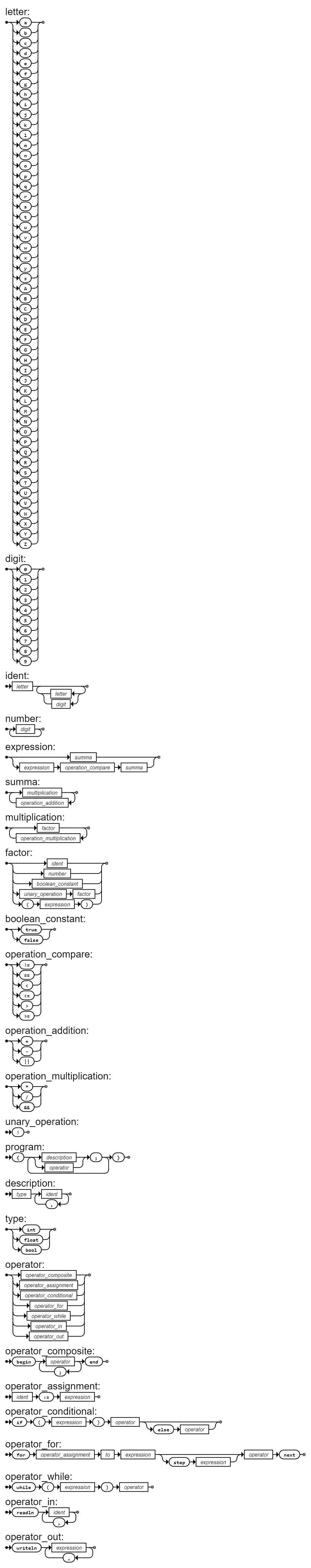


Рисунок 3.1 – Грамматика в виде диаграммы Вирта

# Спецификация основных процедур и функций

Для реализации нашего программного продукта необходимы следующие части

## Лексический анализатор

Задачи лексического анализатора:

* выполняет «сборку» лексем из отдельных символом входного текста. Лексемы — неделимые единицы программы — имена, числа, зарезервированные слова, разделители и знаки операций, которые могут состоять в том числе и из нескольких символов;
* удаляет комментарии и пробельные символы (пробел, табуляций, конец строки).

Исходная грамматика предусматривает 16 лексем, соответствующих ключевым словам (begin, bool, else, end, false, float, for, if, int, next, readln, step, to, true, while, writeln), лексему идентификатора, 5 лексем, соответствующих числам (двоичное, восьмиричное, десятичное, шестнадцатиричное, дробное), 20 лексем разделителей (‘(’, ‘)’, ‘!=’, ‘==’, ‘<’, ‘<=’, ‘>’, ‘>=’, ‘+’, ‘-’, ‘||’, ’\*‘,’/‘,’&&‘,’!‘,’{‘,’}‘,’:=‘,’;’), а также были добавлена лексема, к которой определяются не распознанные наборы символов и лексема маркера конца ввода.

Для работы лексического анализатора также используется модуль, задача которого состоит в чтении исходного содержимого файла программы и хранения его составных частей для возможности подсветки ошибок.

Диаграмма классов приведена на рисунке [4.1](#fig:lexer_classes)

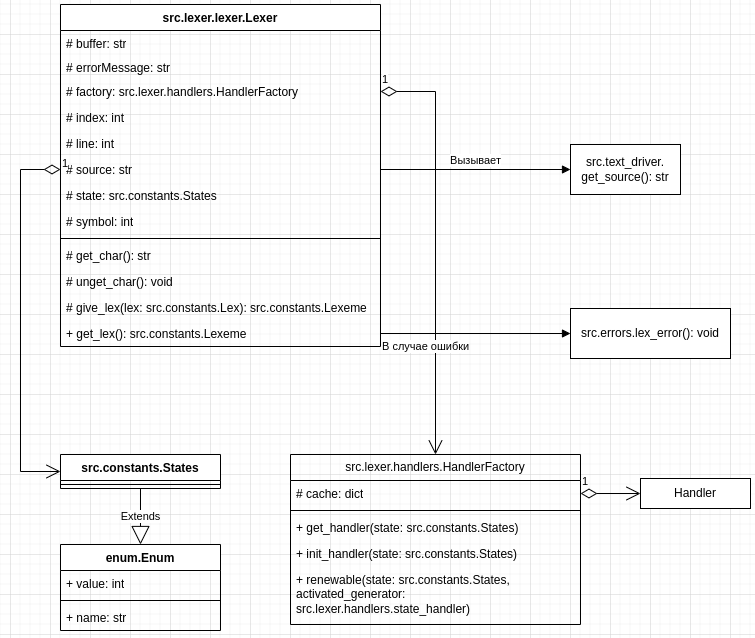


Рисунок 4.1 – Диаграмма классов

### Класс Lexer

Реализует класс лексического анализатора. Входные данные отсутствуют.

Перед началом работы требует вызов функции get\_source драйвера исходного текста.

Поля: - buffer: накапливающийся буфер лексемы; - errorMessage: сообщение об ошибке, использующееся в конце разбора неопределённой лексемы; - factory: объект класса HandlerFactory, выдаёт соответствующий генератор обработки текущего состояния; - index: индекс символа в строке source; - source: исходный текст; - state: текущее состояние.

Использует класс HandlerFactory

Методы:

* get\_char: возвращает очередной символ исходного текста;
* unget\_char: возвращается к предыдущему символу;
* give\_lex: принимает тип лексемы, выдаёт объект Lexeme;
* get\_lex: генератор, выдаёт очередную лексему.

В случае ошибки вызывается метод lex\_error.

Лексический анализатор работает на базе конечного автомата, у него существует множество состояний, значения которых описываются классом src.constants.States.

для каждого из них, за исключением STATE\_NULL существуют обработчики.

Обработчики состояний:

* state\_start\_handler;
* state\_identificator\_handler;
* state\_separator\_equals\_handler;
* state\_separator\_lt\_handler;
* state\_separator\_gt\_handler;
* state\_separator\_or\_handler;
* state\_separator\_and\_handler;
* state\_separator\_assignment\_handler;
* state\_separator\_comment\_handler;
* state\_separator\_not\_handler;
* state\_number\_bin\_handler;
* state\_letter\_b\_handler;
* state\_letter\_d\_hander;
* state\_letter\_e\_hander;
* state\_letter\_h\_handler;
* state\_letter\_o\_handler;
* state\_number\_oct\_handler;
* state\_number\_dec\_handler;
* state\_number\_hex\_handler;
* state\_fractional\_handler;
* state\_number\_order\_handler;
* state\_er\_handler.

Обработчик состояния это генератор, который принимает символ и возвращает либо новое состояние, либо лексему и новое состояние, либо сообщение об ошибке и новое состояние.

## Синтаксический анализатор

Синтаксический анализатор также является и семантическим анализатором.

Диаграмма классов приведена на рисунке [4.2](#fig:parser_classes)

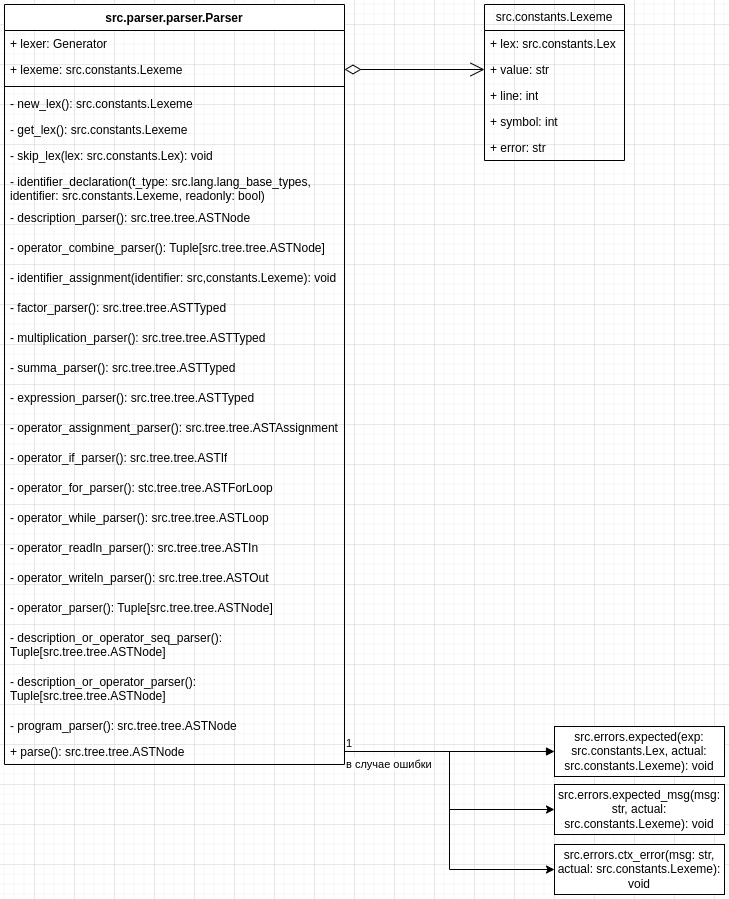


Рисунок 4.2 – Диаграмма классов

Для инициализации объекта класса необходим объект лексического анализатора.

Методы: - new\_lex: запрашивает у лексического анализатора новую лексему и возвращает её; - get\_lex: возвращает текущую лексему; - skip\_lex: принимает тип лексемы, вызывает метод new\_lex и проверяет тип новой лексемы на соответствие переданному агрументу, в случае несоответствия вызывает функцию expected.

Все методы, заканчивающиеся на “\_parser” распознают соответствующий названию метода блок программы.

* description\_parser: распознаёт объявление переменной;
* operator\_combine\_parser: распознаёт комбинированный оператор;
* factor\_parser: парсер значения (число или переменная);
* multiplication\_parser: парсер слагаемого;
* summa\_parser: парсер множителя;
* expression\_parser: парсер выражения;
* operator\_assignment\_parser: парсер оператора присваивания;
* operator\_if\_parser: парсер условного оператора;
* operator\_for\_parser: парсер цикла со счётчиком;
* operator\_while\_parser: парсер цикла с предусловием;
* operator\_readln\_parser: парсер оператора ввода;
* operator\_writeln\_parser: парсер оператора вывода;
* operator\_parser: парсер оператора;
* description\_or\_operator\_seq\_parser: парсер последовательности объявление или операторов;
* description\_or\_operator\_parser: парсер объявления или оператора;
* program\_parser: парсер программы;
* parse: метод парсинга исходного кода всей программы.

В контексте синтаксического анализатора приведённые выше методы, заканчивающиеся на “\_parser” не требуют аргументов.

Также на диаграмме можно обнаружить ещё несколько неописанных методов, но о них и о возвращаемых значениях приведённых выше методов в разделе семантического анализатора.

## Семантический анализатор

Расширенная диаграмма классов приведена на рисунке [4.3](#fig:semantic_classes).

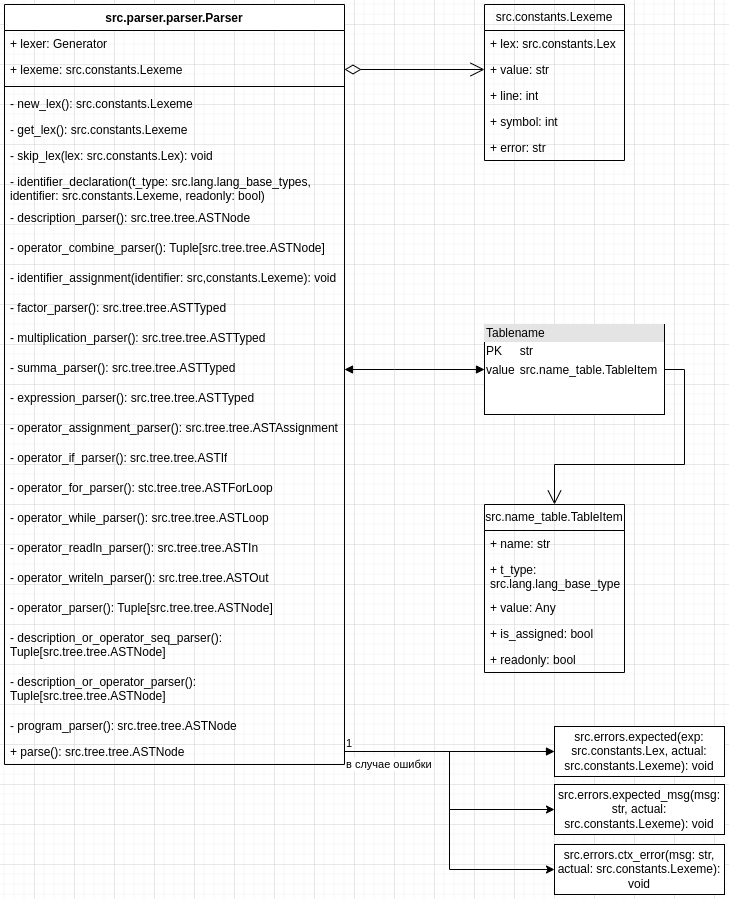


Рисунок 4.3 – Диаграмма классов

Рассмотрим парсер более подробно:

Все методы, заканчивающиеся на “\_parser” возвращают узел дерева разбора, таким образом, что в методе “parse” мы получаем полноценное дерево всей программы.

Метод identifier\_declaration заносит идентификатор в текущую область видимости, принимает на вход: тип данных, асоциированный с переменной, лексему идентификатора, логическое значение, является ли переменная переменной только для чтения.

Метод identifier\_assignment находит идентификатор в таблице имён и устанавливает поле is\_assigned в значение истины. Входные данные: объект лексемы.

Диаграмма классов абстрактного синтаксического дерева приведена на рисунке [4.4](#fig:ast).

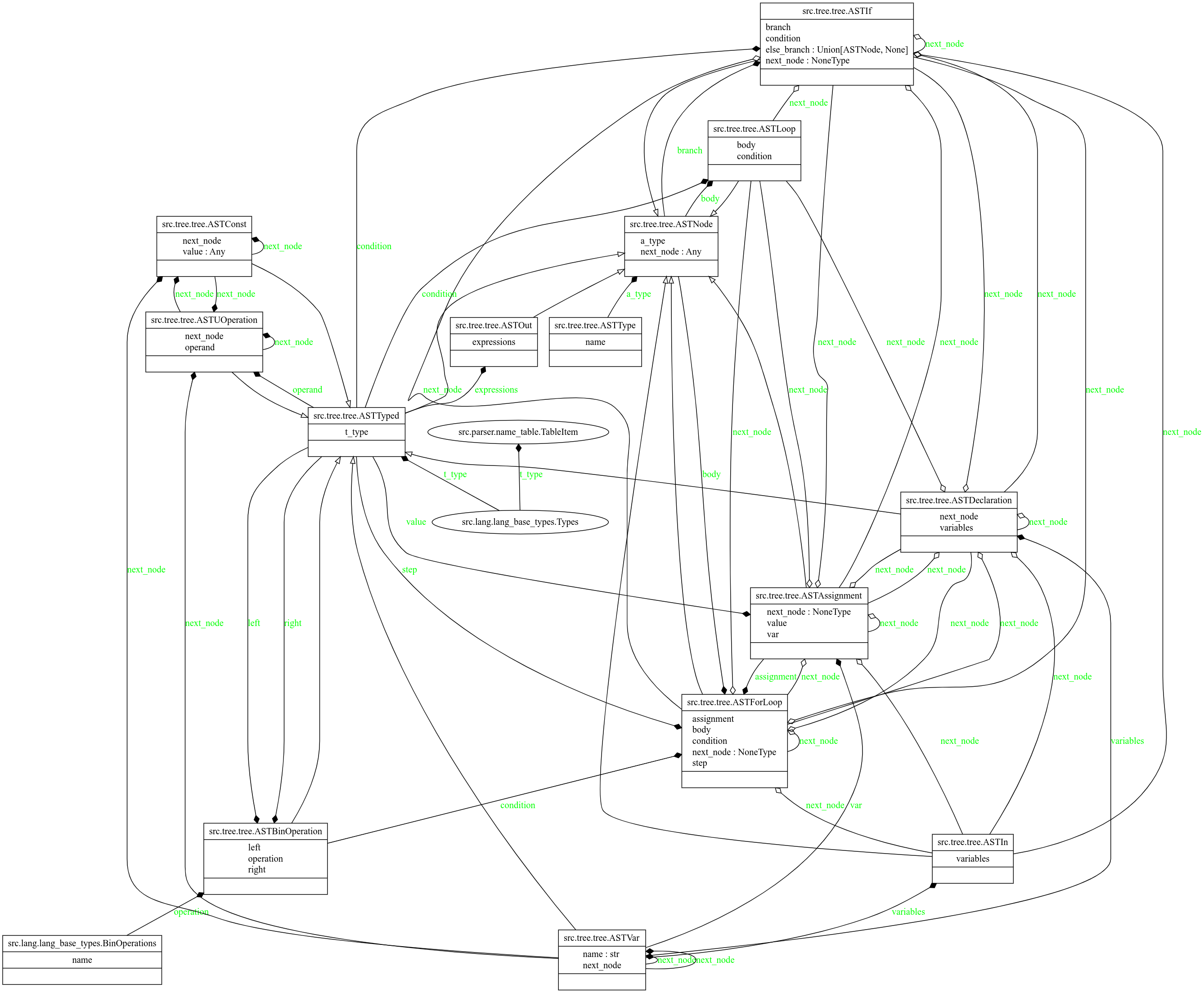


Рисунок 4.4 – Диаграмма классов абстрактного синтаксического дереваа

У языка строгая типизация, поэтому некоторые методы отвечают за проверку типов.

Метод operator\_assignment\_parser проверяет, объявлена ли переменная, не является ли переменная переменной только для чтения, а также соответствие типа переменной и присваемого выражения, после чего вызывает метод identifier\_assignment.

Метод identifier\_declaration вызывает метод identifier\_declaration, который в случае повторного объявления сообщит об ошибке.

Методы factor\_parser, multiplication\_parser, summa\_parser, expression\_parser проверяют на соответствие типов подвыражений слева и справа относительно лексемы операции в соответствии с правилами, определёнными спецификацией языка (см. Приложение 1).

Метод operator\_for\_parser порождает новую область видимости, в которой объявляет переменную для счётчик целочисленного типа.

## Драйвер исходного текста

Функция src.text\_driver.setup\_source: принимает в качестве аргумента путь к файлу с исходным кодом программы на модельном языке, проверяет расширение файла, вызывает функцию read\_file и устанавливает значение переменной is\_set в истину.

Функция src.text\_driver.read\_file: принимает в качестве аргумента путь к файлу, считывает файл, заменяя табуляцию на четыре проблела, заполняет массив \_lines и возвращает содержимое файла в качестве строки.

Функция src.text\_driver.highlight: принимает номер строки и номер символа в строке, подсвечивает данное место.

Функция src.text\_driver.get\_source: возвращает значение переменной \_source.

Функция get\_filename: возвращает значение переменной \_filename.

## Сообщения об ошибках

Функция src.errors.error: принимает на вход сообщение об ошибке, номер строки, номер символа в строке, код завершения программы. Печатает ошибку, вызывает функцию src.text\_driver.highlight драйвера исходного текста и завершает работу программы с соответствующим кодом завершения.

Функция src.errors.lex\_error: принимает объект класса лексемы, формирует сообщение о лексической ошибке, вызывает функцию src.errors.error.

Функция src.errors.expected: принимает тип ожидаемой лексемы и объект фактической лексемы, формирует сообщение о синтаксической ошибке и вызывает функцию src.errors.error.

Функция src.errors.expected\_msg: аналогично src.errors.expected, только вместо типа ожидаемой лексемы принимает сообщение об ошибке.

Функция src.errors.ctx\_error: принимает сообщение об ошибке и объект лексемы, формирует сообщение о контекстной ошибке и вызывает функцию src.errors.error.

## Трансляторы

Функции src.python\_translator.translator.translate и src.cpp\_translator.translator.translate: принимает корневой узел абстрактного синтаксического дерева src.tree.tree.ASTNode, возвращает строку с программой на целевом языке.

Функции src.python\_translator.translator.translate\_expression и src.cpp\_translator.translator.translate\_expression: принимает узел абстрактного синтаксического дерева src.tree.tree.ASTTyped, соответствующий выражению и возвращает строку с его эквивалентом на целевом языке с расставленными скобками.

Функции src.python\_translator.translator.translate\_operator и src.cpp\_translator.translator.translate\_operator принимают узел абстрактного синтаксического дерева src.tree.tree.ASTNode, соответствующий оператору и возвращает строку с кодом программы на целевом языке, соответствующую этому оператору.

# Структурная организация данных

Самые частоиспользуемые типы данных, это класс лексемы и класс типа лексемы, они приведены на рисунке [5.1](#fig:lexeme).

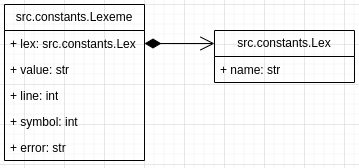


Рисунок 5.1 – Класс лексемы и класс типа лексемы

Класс src.constants.Lex является наследником класса Enum и его экземпляр может иметь одно из следующих мнемонических значений (KEYWORD\_BEGIN, KEYWORD\_BOOL, KEYWORD\_ELSE, KEYWORD\_END, KEYWORD\_FALSE, KEYWORD\_FLOAT, KEYWORD\_FOR, KEYWORD\_IF, KEYWORD\_INT, KEYWORD\_NEXT, KEYWORD\_READLN, KEYWORD\_STEP, KEYWORD\_TO, KEYWORD\_TRUE, KEYWORD\_WHILE, KEYWORD\_WRITELN, IDENTIFIER, NUMBER\_BIN, NUMBER\_OCT, NUMBER\_DEC, NUMBER\_HEX, NUMBER\_FRACTIONAL, SEPARATOR\_AND, SEPARATOR\_ASSIGNMENT, SEPARATOR\_DIVISION, SEPARATOR\_EQUALS, SEPARATOR\_GT, SEPARATOR\_GTE, SEPARATOR\_LEFT\_BRACKET, SEPARATOR\_LEFT\_FIGURE\_BRACKET, SEPARATOR\_LT, SEPARATOR\_LTE, SEPARATOR\_MINUS, SEPARATOR\_MULTIPLICATION, SEPARATOR\_NOT, SEPARATOR\_NOT\_EQUALS, SEPARATOR\_OR, SEPARATOR\_PLUS, SEPARATOR\_RIGHT\_BRACKET, SEPARATOR\_RIGHT\_FIGURE\_BRACKET, SEPARATOR\_SEMICOLON, SEPARATOR\_COMMA, UNRESOLVED, EOF).

Класс src.constants.States также является наследником класса enum и имеет следующие значения (ER, START, IDENTIFIER, NUMBER\_BIN, NUMBER\_OCT, NUMBER\_DEC, NUMBER\_HEX, NUMBER\_ORDER, FRACTIONAL, LETTER\_B, LETTER\_D, LETTER\_E, LETTER\_H, LETTER\_O, SEPARATOR\_EQUALS, SEPARATOR\_OR, SEPARATOR\_AND, SEPARATOR\_ASSIGNMENT, SEPARATOR\_NOT, SEPARATOR\_LT, SEPARATOR\_GT, SEPARATOR\_COMMENT, STATE\_NULL).

Разделители лексем представлены двумя кортежами.

BASE\_SEPARATORS включает пробельный символ, табуляцию и перенос строки. SEPARATORS включает BASE\_SEPARATORS и следующие символы: “(”, “)”, “!”, “=”, “<”, “>”, “+”, “-”, “|”, “\*“,”/“,”&“,”{“,”}“,”:“,”;“,”,“.

Каждый из генераторов в файле src.lexer.handlers, заканчивающийся на “\_handler” принимает символ типа str, возвращает либо новое состояние типа src.constants.States, либо кортеж из пары (src.constants.Lex, src.constants.States), что означает распознанную лексему, либо список из пары (str, src.constants.States), что означает, что при распознавании произошла ошибка.

существует хэш-таблица HANDLERS, ключи которой соответствуют всем возможным значениями src.constants.States, а значения соответствуют ссылкой на генератор-обработчик данного состояния.

Семантический анализатор использует таблицу имён, описываемую диаграммой на рисунке [5.2](#fig:name_table).

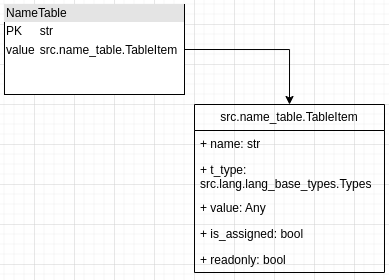


Рисунок 5.2 – Таблица имён

В языке предусмотрены три типа (целочисленный, вещественный и логический), принадлежность к типу определяется классом src.lang.lang\_base.types.Types, являющимся наследником от Enum и имеющий значения (int, float, bool).

Предусмотрены 13 различных операций, одна из которых появляется только в ходе оптимизации. Для их описания используется класс BinOperations, наследующийся от Enum и имеющий значения (sum, diff, mul, div, alt, con, eq, neq, gt, gte, lt, lte, mod).

Абстрактное синтаксическое дерево имеет разные виды узлов, для опеределения типа узла используется класс src.tree.tree.ASTType, наследующийся от Enum и имеющий следующие значения (ASSIGNMENT, IF, Loop, IN, OUT, DECL, U\_OP, BIN\_OP, CONST, VAR, ForLoop).

Диаграмма классов абстрактного синтаксического дерева представлена на рисунках [5.3](#fig:ast_1) - [5.4](#fig:ast_2).

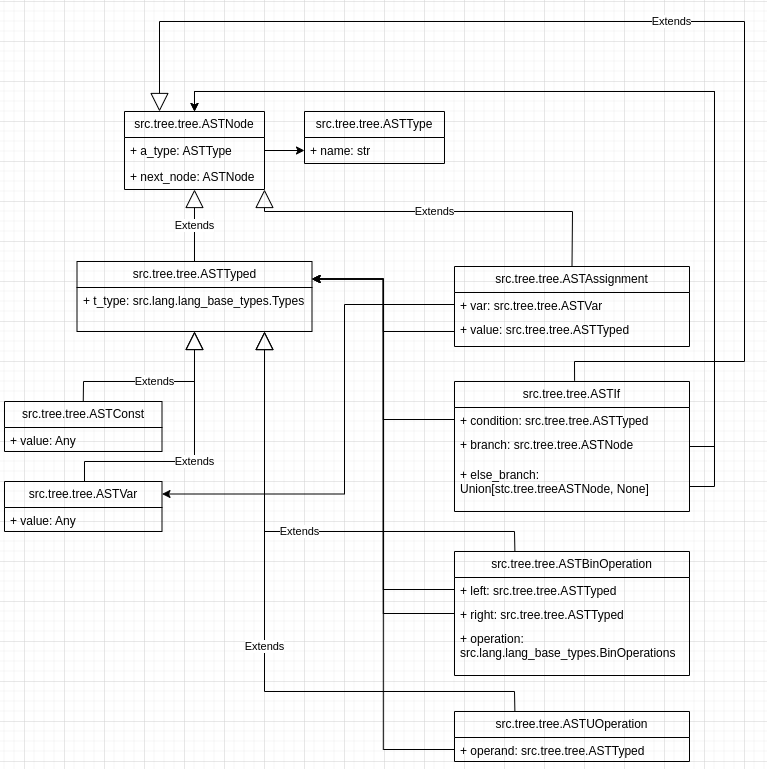


Рисунок 5.3 – Абстрактное синтаксическое дерево, часть 1

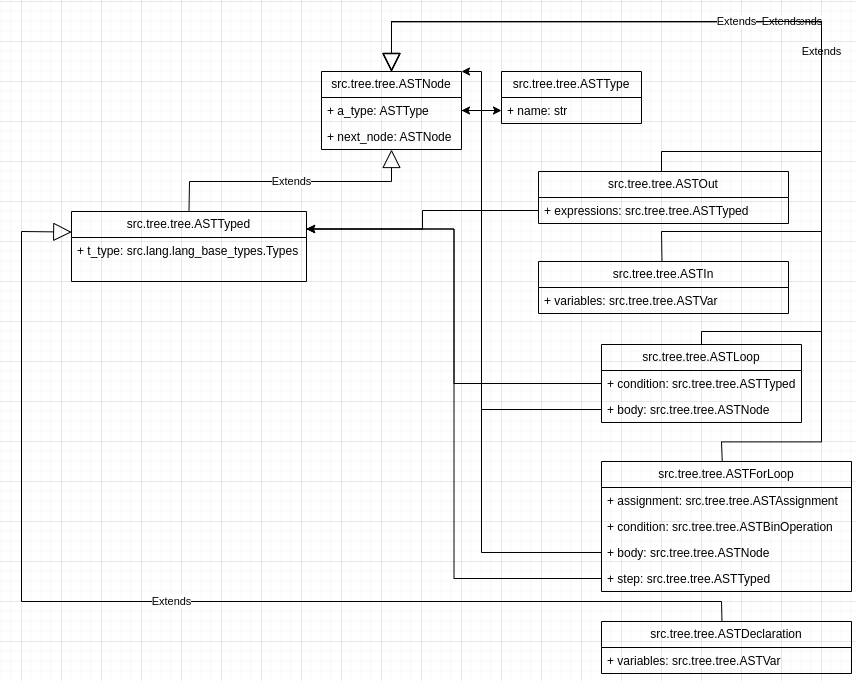


Рисунок 5.4 – Абстрактное синтаксическое дерево, часть 2

В файле src.text\_driver.py используются следующие данные: - \_source: str - исходный код программы; - \_lines: List[str] - массив строк исходного кода программы; - \_is\_set: bool - логическое значение, соответствующее готовности программы к дальнейшей обработке; - \_filename: str - имя обрабатываемого файла.

## Спецификация входных данных

Работа с данным программным средством производится через командную строку.

Входные данные: текстовый файл с расширением “.leo”, опциональный флаг “-v”, опциональный флаг “-h”, опциональный флаг “-t” и одно из (“py”, “cpp”), опциональный флаг “-o” и имя выходного файла.

## Спецификация выходных данных

Выходные данные: Версия программного средства, справка о работе командного средства, файл с оттранслированной программой на язык c++, файл с оттранслированной программой на язык python, оттранслированная программа на язык c++, оттранслированная программа на язык python, сообщение об ошибке.

# Разработка алгоритма решения задачи

Задача будет решаться в два прохода.

Процедуры перед началом прохода:

* подготовка исходных данных, вызов функции setup\_source;
* создание объекта класса Lexer;
* создание объекта класса Parser с передачей объекта класса Lexer в качестве аргумента.

Первый проход: - вызов метода parse объекта класса Parser, который будет требовать от лексера новые лексемы.

Второй проход: - Вызов соответствующего метода трансляции (src.python\_translator.translator.translate или src.cpp\_translator.translator.translate) и получений программы на целевом языке.

Процедуры после проходов: - сохранение в файл или печать программы в консоль.

## Укрупненная схема алгоритма программного средства

Укрупненная схема алгорита приведена на рисунках [6.1](#fig:alg1) - [6.3](#fig:alg3).

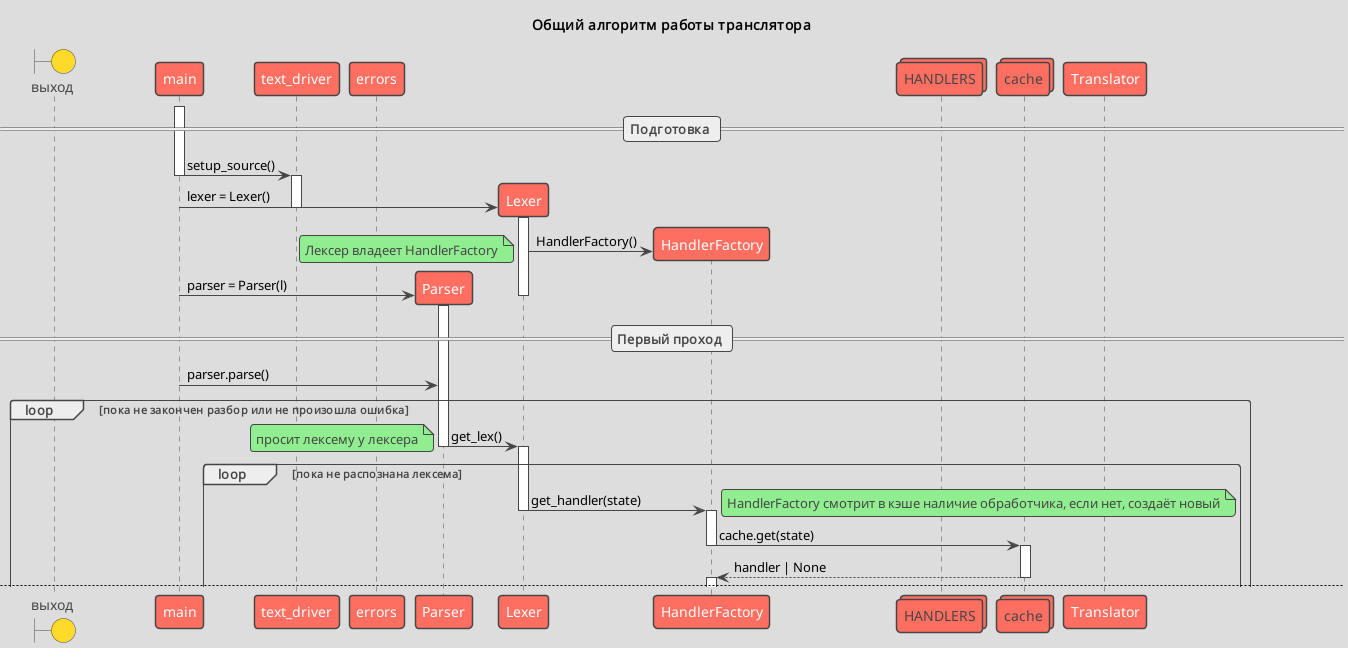


Рисунок 6.1 – Укрупненная схема алгорита, часть 1

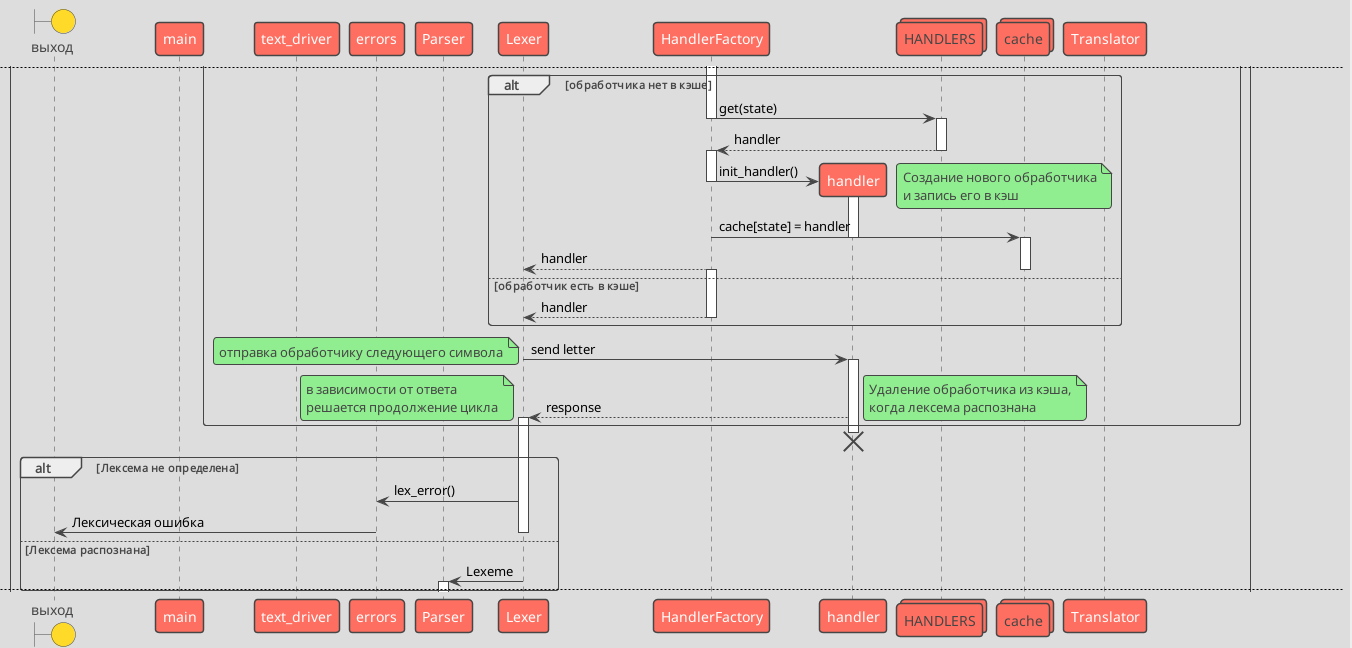


Рисунок 6.2 – Укрупненная схема алгорита, часть 2

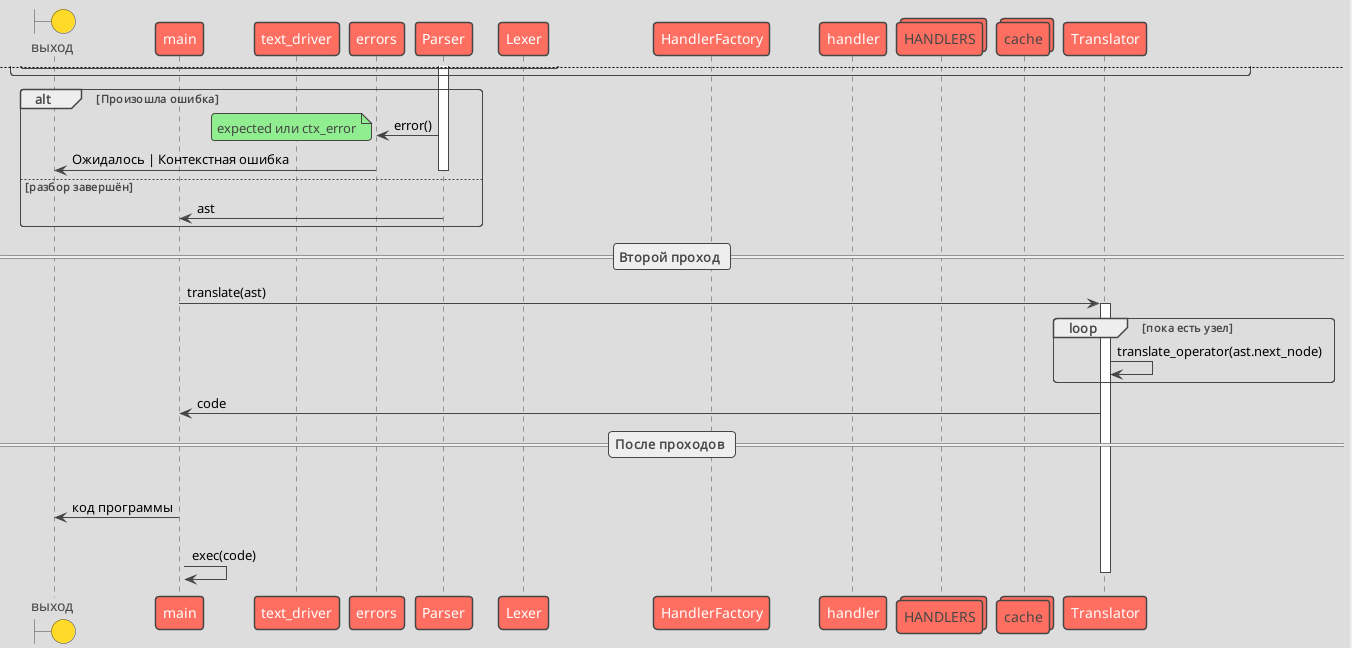


Рисунок 6.3 – Укрупненная схема алгорита, часть 3

## Детальная разработка алгоритмов отдельных подзадач

Перед началом работы программы пользователю необходимо ввести название файла с исходным кодом программы на модельном языке.

Задача инициализации:

Перед началом парсинга была выявлена необходимость в отдельном модуле для чтения данных из файла, а также хранении списка строк для подсветки места, в котором может произойти ошибка.

Инициализация происходит посредством вызова функции setup\_source. Функция проверяет название файла на валидность, вызывает функцию чтения файла, которая заполняет строковое поле \_source с исходным кодом программы и список строк \_lines, предварительно заменяя символ табуляции на 4 пробела для корректной подсветки ошибок.

После этого необходимо создать объект класса Lexer, при инициализации он вызовет функцию get\_source, которая выдаст ошибку, если предварительно не была вызвана функция get\_source. Множество полей объекта класса будет проинициализированно начальными значениями, но особое внимание стоит обратить на создание объекта класса HandlerFactory.

После создания объекта класса лексического анализатора необходимо создать объект класса синтаксического анализатора, передав ему в качестве аргумента созданный лексический анализатор. Синтаксический анализатор интересует метод get\_lex лексического анализатора. Это генератор, оборачиваемый в итератор, для которого по мере необходимости вызывается функция next.

После того, как необходимые объекты созданы можно начинать парсить.

Алгоритм работы get\_lex класса Lexer.

Данный метод работает на базе конечного автомата, при этом используя бесконечный цикл. Для грамматики со сложностью, соответствующей исходной нельзя было обойтись без метода unget\_char, при условии использования такого цикла.

В бесконечном цикле вызывается метод get\_char, результат работы которого записывается в переменную char, это очередной символ исходного текста программы. Если прочитать символ не удалось, генератор отправляет лексему конца файла и завершает свою работу. Далее у поля \_factory вызывается метод get\_handler с параметром текущего состояния. Метод возвращает обработчик для переданного состояния. get\_handler сначала ищет существующий объект обработчика в кэше, реализованном хэш-таблицей и если его там не оказалось, создаётся новый с помощью хэш-таблицы HANDLERS. Созданный генератор инициализируется и оборачивается в генератор-делегатор renewable, задача которого состоит в том, что если новое состояние, которое вернёт обработчик будет отличаться от того, которое он обрабатывает, в кэш запишется новый объект такого генератора, для того, чтобы его состояние снова стало исходным. Обработчики хоть и обрабатывают состояния, однако сами могут иметь свои подсостояния, например обработчик состояния символа двойного равенства имеет два состояния, в первом он ожидает второй символ ‘=’, во втором он выдаёт лексему, а первый символ ‘=’ обрабатывается в состоянии START. Каждый обработчик отдаёт лексему только тогда, когда получит символ, не соответствующей ей, после чего необходимо будет выполнить метод unget\_char. Например при обработке “==”. Лексема символа равенства будет выдана после обработки символа проблела, после чего в цикле должен считываться следующий символ.

После получения нужного обработчика ему отправляется следующий символ. Если ответ от обработчика это новое состояние, то состояние меняется на него, если это кортеж из типа лексемы и нового состояния, то необходимо отдать лексему, а если это список из сообщения об ошибке и нового состояния, то сообщение запоминается и генератор переходит в состояние обработки ошибки, ожидая символ разделитель.

Если символ не принадлежит базовым разделителям (пробел, табуляция, перенос строки, он накапливается в поле buffer). Если обработчик отдал лексему идентификатора, то необъодимо проверить, является ли она ключевым словом. Одновременно с этим высчитываются номера строк и символа в строке для того, чтобы ассоциировать с ними лексему.

Метод give\_lex отдаёт лексему с учётом накопленного буфера и всех ассоциированных значений с лексемой (тип, значение, номер строки, номер символа в строке). Если тип лексемы UNRESOLVED, вызывается функция lex\_error. Иначе get\_lex отдаёт лексему, сохраняя своё состоянии и поток управления возвращается к тому, кто её запросил, то есть синтаксическому анализатору.

Алгоритм работы класса Parser.

Класс одновременно анализирует программу на синтаксическую и семантическую корректность, а также строит абстрактное синтаксическое дерево.

Для реализации трансляции на Python возникла необходимость в создании переменной типа только для чтения. Дело в цикле for, в Python ему соответствует конструкция “for i in range”, это не позволяет изменять переменную, ведь каждую итерацию её значение определяется заново, а новая область видимости необходима для того, чтобы счётчик нельзя было использовать вне цикла. Для помощи в проверки программы на семантическую корректность была создана таблица имён, это хэш-таблица, элемент которой класс TableItem.

Метод parse начинает парсинг, реализующий алгоритм рекурсивного спуска. Каждый метод, заканчивающийся на “\_parser” соответствует правилам в грамматике языка. Метод parse\_program парсит конструкцию, соответствующую правилу “”, description\_or\_operator\_parser конструкцию, соответствующую правилу “( | )” и далее по аналогии. Каждый метод возвращает узел абстрактного синтаксического дерева, реализованного на базе односвязного списка, но имеющим ответвления.

Стоит отметить следующие детали:

description\_parser (парсер оператора объявления) вызывает метод indentifier\_declaration, который вызывает функцию new\_name, которая создаёт запись о переменной в таблице имён, а в случае, если переменная там уже есть выдаёт контекстную ошибку.

operator\_combine\_parser (парсер комбинированного оператора) возвращает два узла, начальный и конечный, так как множество операторов собираются в односвязный список.

operator\_assignment\_parser (парсер оператора присваивания) проверяет идентификатор, к которому производится присваивание на наличие в таблице имён, если его там нет, выдаёт контекстную ошибку о том, что переменная не объявлена, если запись найдена, но переменная доступна только для чтения, выдаёт контекстную ошибку, если тип присваемого выражения отличается от типа переменной, также выдаёт ошибку.

operator\_readln\_parser (парсер оператора ввода) также осуществляет проверки на предварительное определение переменной и на принадлежность переменной к классу переменный только для чтения.

operator\_for\_parser (парсер цикла со счётчиком) самый сложный, ведь ему необходимо объявить переменную счётчик с параметром только для чтения, проверить тип переменной, присваемого выражения, а также выражению конца цикла и выражению после ключевого слова “step” на соответствие целочисленному типу.

expression\_parser, summa\_parser, multiplication\_parser, factor\_parser проверяют на соответствие типов подвыражений слева и справа от лексемы типа операции на предмет соответствия спецификации языка.

После того, как программа распознана, а дерево построено производится одна единственная оптимизация, выражение вида “i / j \* j == i” будет заменено на выражение вида “i % j == 0” на уровне синтаксического дерева, при условии, что i и j объявлены как переменные целочисленного типа.

После того, как оптимизация была проведена, выполняется трансляция. функция translate импортирует стандартную библиотеку в случае с c++, и для обоих целевых языков формирует объявление функции main. Далее движение происходит напрямую по полям next\_node верхнеуровневых узлов. Для каждого узла вызывается функция translate\_operator, которая переводит смысл оператора на целевой язык. функция translate\_expression вызывается функцией translate\_operator и переводит выражение на целевой язык, расставляя скобки.

После трансляции в зависимости от запроса пользователя программа либо выполняется «на месте» при помощи команды exec, либо выводится в консоль, либо сохраняется в файл.

# Установка и эксплуатация программного средства

## Условия эксплуатации

Программный продукт требуется использовать на поддерживающихся устройствах в их диапазоне поддерживающихся температур эксплуатации и хранения.

## Требования к составу и параметрам технических средств

Для пользовательских устройств: Пользовательское устройство должно иметь одну из следующих операционных систем и соответствующие для неё требования.

1. Windows:

* windows 10 или более поздней версии либо Windows Server 2016 или более поздней версии;
* процессор Intel Pentium 4, а также его аналоги или более поздние их версии с поддержкой SSE3.

1. MacOS:

* macOS High Sierra 10.13 или более поздней версии.

1. Linux:

* ubuntu 18.04 (64-разрядная версия) или более поздней версии;
* debian 10 или более поздней версии, openSUSE 15.2 или более поздней версии;
* fedora Linux 32 или более поздней версии.

1. Android:

− android 7.0 Nougat или более поздней версии.

Также для работы системы требуются:

* от 4-х мегабайт свободного дискового пространства;
* от 250 мегабайт оперативной памяти.

## Установка программного средства

Для установки программного средства необходимо установить Python версии 3.8 и выше, после чего скачать исходный код программы.

# Работа с программным средством

Программное средство предусматривает следующие сценарии использования:

1. Интерпретация программы на языке Leo.
2. Трансляция программы на языке Leo.
3. Получение справки об использовании.
4. Получение информации о версии программной системы.

Программа, реализующая программный интерфейс называется leo.py в каталоге course.

## Интерпретация программы на языке Leo.

Для данного сценария использования предусматривается следующая конструкция обращения “python leo.py название\_программы.leo”.

При отсутствии файла или отличном от “.leo” расширении будет выведено сообщение как на рисунке [8.1](#fig:invalid_path).

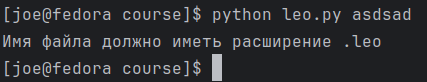


Рисунок 8.1 – Некорректный файл

В случае успеха программа начнёт своё выполнение. Имеют место ошибки, связанные с некорректным вводом данных для самой программы, например ввод строки для ожидаемого программой целого числа. Они будут обработаны самим Python, например как на рисунке [8.2](#fig:invalid_input).

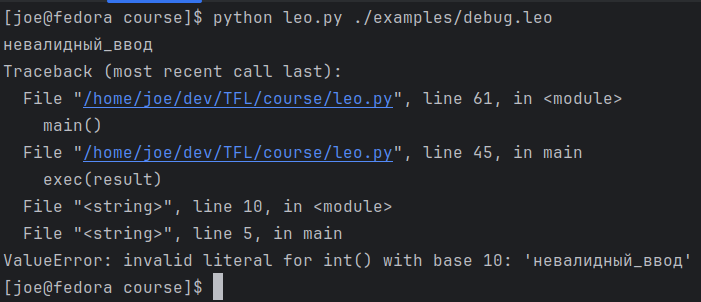


Рисунок 8.2 – Некорректный ввод

## Трансляция программы на языке Leo

Для данного сценария использования предусматривается следующая конструкция обращения “python leo.py название\_программы.leo -t {py, cpp} -o {название\_выходной\_программы, std}”, где “{py, cpp}” означает одно из “py”, “cpp”. “py” соответствует трансляции на Python, “cpp” на c++.

При отсутствии файла или отличном от “.leo” расширении будет выведено сообщение как на рисунке [8.1](#fig:invalid_path).

При выборе варианта std программа будет выведена в консоль.

Иначе программа на целевом языке будет записана в файл.

## Получение справки об использовании

Справка будет выведена при обращении “python leo.py -h” или при неправильном вводе.

Результат вывода справки изображён на рисунке [fig:help].

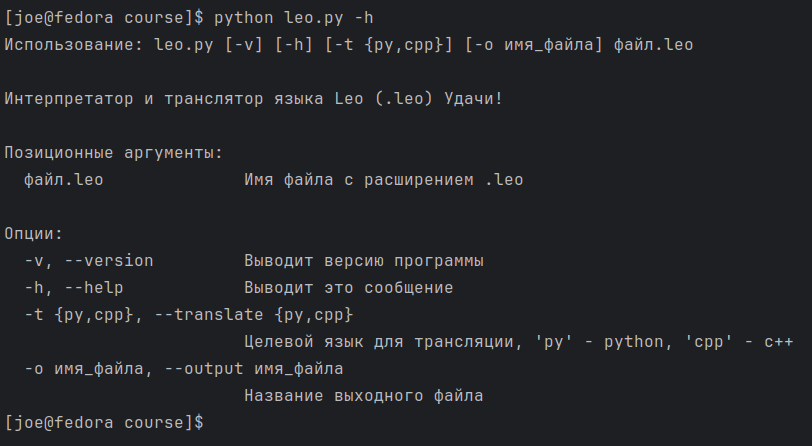


Рисунок 8.3 – Справка

## Получение версии программной системы

Версия будет выведена при обращении “python leo.py -v”.

Результат вывода справки изображён на рисунке [fig:version].

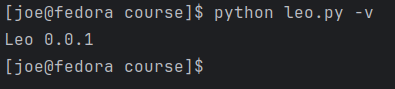


Рисунок 8.4 – Получение версии программной системы

# Заключение

В результате выполнения данной курсовой работы была решена задача создания языка программирования на базе исходной грамматики.

Здесь я напишу, что поставленные задачи (из раздела выше) были решены

Разработанная система способна интерпретировать программы на языке Leo, а также транслировать их на языки Python и c++. Система позволит писать общий код для двух различных языков программирования, что значительно ускорит скорость разработки несложных вычислительных программ и использовать код на целевых языках в зависимости от наличия их компиляторов/интерпретаторов на конкретной машине.

Система имеет потенциал для добавления для различных оптимизаций, статического и динамического анализа кода, добавлению поддержки трансляции на другие языки программирования, написанию виртуальной машины и генерации кода для неё или непосредственно для соответствующих архитекрут процессоров. Также исходная грамматика поддаётся модификации и нуждается в некоторых коррекциях, а также поддержки пользовательских функций, модулей, конвертации типов, пользовательских структур данных и разбиению кода программы на несколько файлов.

# Список использованных источников

1. Сергей Свердлов Конструирование компиляторов 2005. - ISBN 978-3-659-71665-2.
2. Никлаус Вирт Построение компиляторов? Москва 2010. - ISBN 978-5-94074-585-3.
3. Ахо, А.В. Компиляторы: принципы, технологии и инструменты / А. Ахо, Р. Сети, Д. Ульман; перевод с англ. И.В. Красикова и др. - М.: Вильямс, 2001. - 767 с.: ил.; 24 см. - Библиогр.: с. 742-763. - Предм. указ.: 764-767. - 5000 экз. - ISBN 5-8459-0189-8 (в пер.).
4. Власенко, А.В. Теория языков программирования и методы трансляции: учеб. пособие / А.В. Власенко, В.И. Ключко; М-во образования и науки РФ, ГОУ ВПО «Кубан. гос. технол. ун-т». - Краснодар: Изд-во КубГТУ, 2004. - 119 с.: ил.; 21 см. - Библиогр.: с. 118. - 75 экз. - ISBN 5-8333-0176-9.
5. Гавриков, М.М. Основы конструирования компиляторов: учеб. пособие / М.М. Гавриков, А.Н. Иванченко, Д.В. Гринченков; М-во общ. и проф. образования РФ, Новочеркас. гос. техн. ун-т. – Новочеркасск: НГТУ, 1997. - 80 с.: ил.; 20 см. - Библиогр.: с. 79. – 75 экз. - ISBN 5-88998-059-9.
6. Гордеев, А.В. Системное программное обеспечение: учеб. для вузов / А. Ю. Молчанов. - 3-е изд. - СПб.: Питер, 2010. - 398 с.: ил. - (Учебник для вузов). - Указ. лит.: с. 387-390. - Алф. указ.: с. 391-397. - ISBN 978-5-49807-153-4.
7. Ишакова, Е.Н. Теория языков программирования и методов трансляции: учебное пособие / Е.Н. Ишакова. – Оренбург: ИПК ГОУ ОГУ, 2007. – 137 с. - ISBN 978-5-7410-0712-9.
8. Ишакова, Е.Н. Разработка компиляторов: Методические указания к курсовой работе / Е.Н. Ишакова. - Оренбург: ГОУ ОГУ, 2005. – 50 с.
9. Карпов, В.Э. Классическая теория компиляторов: учеб. пособие / В.Э. Карпов; М-во образования РФ, Моск. гос. ин-т электрон. и математики (техн. ун-т). - М.: МГИЭМ, 2002. - 78 с : ил.; 20 см. - Библиогр.: с. 78. - 150 экз. - ISBN 5-230-16344-5.
10. Компаниец, Р.И. Системное программирование: основы построения трансляторов: учеб. пособие для высших и средних учебных заведений / Р.И. Компаниец, Е.В. Маньков, Н.Е. Филатов. - СПб.: Корона принт, 2000. - 254, [1] с.: ил.; 23 см. - Библиогр.: с. 255. - 3000 экз. - ISBN 5-7931-0124-1.
11. Мозговой, М.В. Классика программирования: алгоритмы, языки, автоматы, компиляторы. Практический подход / М.В. Мозговой. – СПб.: Наука и техника, 2006. - 320 с.: ил.; 24 см. - 3000 экз. - ISBN 5-94387-224-8.
12. Пратт, Т. Языки программирования: разработка и реализация / Т. Пратт, М. Зелковиц; пер. с англ. - СПб.: Питер принт, 2002. - 688 с.: ил.; 24 см. - Библиогр.: с. 669-674. - Алф. указ.: с. 675-688. - Загл. и авт. ориг.: Programming languages / Terrence W. Pratt, Marvin V. Zelkowitz. - 4000 экз. - ISBN 5-318-00189-0 (в пер.).

# Приложение А — Текст программы

Исходный код программы можно загрузить по ссылке [исходный код программы](https://github.com/Papr1ka/TFL.git).

Листинг 11.1 – leo.py

import argparse  
import pathlib  
  
import src  
  
  
  
class RussianHelpFormatter(argparse.HelpFormatter):  
 def add\_usage(self, usage, actions, groups, prefix=None):  
 if prefix is None:  
 prefix = 'Использование: '  
 return super(RussianHelpFormatter, self).add\_usage(  
 usage, actions, groups, prefix)  
def main():  
 DESCRIPTION = """Интерпретатор и транслятор языка Leo (.leo) Удачи!"""  
  
 parser = argparse.ArgumentParser(description=DESCRIPTION, formatter\_class=RussianHelpFormatter, add\_help=False)  
 parser.\_positionals.title = "Позиционные аргументы"  
 parser.\_optionals.title = "Опции"  
 parser.add\_argument('-v', '--version', action='version',  
 version=f'Leo {src.\_\_version\_\_}', help="Выводит версию программы")  
 parser.add\_argument('-h', '--help', action='help', default=argparse.SUPPRESS,  
 help='Выводит это сообщение')  
 parser.add\_argument("target", type=pathlib.Path, help="Имя файла с расширением .leo", metavar="файл.leo")  
 parser.add\_argument("-t", "--translate", type=str, choices=['py', 'cpp'], default="py",  
 help="Целевой язык для трансляции, 'py' - python, 'cpp' - c++")  
 parser.add\_argument("-o", "--output", type=str, help="Название выходного файла", metavar="имя\_файла")  
 namespace = parser.parse\_args()  
  
 try:  
 src.setup\_source(namespace.target)  
 except ValueError as E:  
 print(E.args[0])  
 exit(4)  
 except FileNotFoundError as E:  
 print("Файл не найден")  
 exit(5)  
  
 lexer = src.Lexer()  
 parser = src.Parser(lexer)  
 ast = parser.parse()  
  
 if namespace.output is None:  
 result = src.py\_translate(ast)  
 exec(result)  
 else:  
 if namespace.translate == "py":  
 result = src.py\_translate(ast)  
 else:  
 result = src.cpp\_translate(ast)  
  
 if namespace.output == "std":  
 print(result)  
 else:  
 with open(namespace.output, "w") as file:  
 file.writelines(result)  
  
 print("Leo: Успешно")  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 main()

Листинг 11.2 – src/text\_driver.py

import os  
import pathlib  
from typing import Union  
  
\_source = ""  
\_lines = []  
\_is\_set = False  
\_filename = "Интерактивный"  
  
  
def read\_file(filename: pathlib.Path) -> str:  
 global \_lines  
 with open(os.path.abspath(os.path.relpath(filename))) as file:  
 \_lines = file.readlines()  
 if not \_lines:  
 \_lines = [""]  
 stream = "".join(\_lines) + " "  
 stream = stream.replace("\t", " ") # замена табов на 4 пробела  
 return stream  
  
  
def highlight(line: int, symbol: int):  
 if len(\_lines) > line - 1:  
 content: str = \_lines[line - 1]  
 if content.endswith("\n"):  
 print(content, end="")  
 else:  
 print(content)  
 for i in range(symbol - 1):  
 print(" ", end="")  
 print("^")  
  
  
def setup\_source(filename: Union[pathlib.Path, str]):  
 global \_source, \_is\_set, \_filename  
 if isinstance(filename, str):  
 filename = pathlib.Path(filename)  
  
 if filename.name.endswith(".leo"):  
 \_source = read\_file(filename)  
 \_filename = filename.name  
 else:  
 raise ValueError("Имя файла должно иметь расширение .leo")  
 \_is\_set = True  
  
  
def get\_source() -> str:  
 if not \_is\_set:  
 raise NotImplementedError("Не вызван setup\_source")  
 return \_source  
  
  
def get\_filename() -> str:  
 return \_filename

Листинг 11.3 – src/errors.py

from .constants import Lex, Lexeme, semantic  
from .text\_driver import highlight, get\_filename  
  
first\_string = 'Файл "{}", Строка {}, Символ {}'  
  
  
def error(message: str, line: int, symbol: int, exit\_code: int):  
 print(first\_string.format(get\_filename(), line, symbol))  
 highlight(line, symbol)  
 print(message)  
 exit(exit\_code)  
  
  
def lex\_error(lex: Lexeme):  
 error(f"Лексическая ошибка: '{lex.value}' - {lex.error}", lex.line, lex.symbol, 1)  
  
  
def expected(exp: Lex, actual: Lexeme):  
 error(f"Синтаксическая ошибка: ожидалось - '{semantic.get(exp)}', получено - {semantic.get(actual.lex)}",  
 actual.line, actual.symbol, 2)  
  
  
def expected\_msg(msg: str, actual: Lexeme):  
 error(f"Синтаксическая ошибка: ожидалось - '{msg}', получено - {semantic.get(actual.lex)}",  
 actual.line, actual.symbol, 2)  
  
  
def ctx\_error(msg: str, lex: Lexeme):  
 error(f"Контекстная ошибка: '{lex.value}' - {msg}",  
 lex.line, lex.symbol, 3)

Листинг 11.4 – src/constants.py

from dataclasses import dataclass  
from enum import Enum  
  
  
class States(Enum):  
 # Рабочие состояния  
 ER = 0  
 START = 1  
 IDENTIFIER = 2  
 NUMBER\_BIN = 3  
 NUMBER\_OCT = 4  
 NUMBER\_DEC = 5  
 NUMBER\_HEX = 6  
 NUMBER\_ORDER = 7  
 FRACTIONAL = 10  
 LETTER\_B = 11  
 LETTER\_D = 12  
 LETTER\_E = 13  
 LETTER\_H = 14  
 LETTER\_O = 15  
  
 # Разделители  
 SEPARATOR\_EQUALS = 17  
 SEPARATOR\_OR = 20  
 SEPARATOR\_AND = 21  
 SEPARATOR\_ASSIGNMENT = 22  
 SEPARATOR\_NOT = 32  
 SEPARATOR\_LT = 33  
 SEPARATOR\_GT = 34  
 SEPARATOR\_COMMENT = 35  
 STATE\_NULL = -1 # вспомогательное состояние, когда буфер накопился, но это не лексема (комментарий)  
  
  
# Границы лексем, не накапливаемые в буфере  
BASE\_SEPARATORS = (  
 " ",  
 "\n",  
 "\t",  
)  
  
# Границы лексем  
SEPARATORS = (  
 \*BASE\_SEPARATORS,  
 "(",  
 ")",  
 "!",  
 "=",  
 "<",  
 ">",  
 "+",  
 "-",  
 "|",  
 "\*",  
 "/",  
 "&",  
 "{",  
 "}",  
 ":",  
 ";",  
 ",",  
)  
  
  
class Lex(Enum):  
 KEYWORD\_BEGIN = 1  
 KEYWORD\_BOOL = 2  
 KEYWORD\_ELSE = 3  
 KEYWORD\_END = 4  
 KEYWORD\_FALSE = 5  
 KEYWORD\_FLOAT = 6  
 KEYWORD\_FOR = 7  
 KEYWORD\_IF = 8  
 KEYWORD\_INT = 9  
 KEYWORD\_NEXT = 10  
 KEYWORD\_READLN = 11  
 KEYWORD\_STEP = 12  
 KEYWORD\_TO = 13  
 KEYWORD\_TRUE = 14  
 KEYWORD\_WHILE = 15  
 KEYWORD\_WRITELN = 16  
  
 IDENTIFIER = 17  
  
 NUMBER\_BIN = 18  
 NUMBER\_OCT = 19  
 NUMBER\_DEC = 20  
 NUMBER\_HEX = 21  
 NUMBER\_FRACTIONAL = 22  
  
 SEPARATOR\_AND = 23  
 SEPARATOR\_ASSIGNMENT = 24  
 SEPARATOR\_DIVISION = 25  
 SEPARATOR\_EQUALS = 26  
 SEPARATOR\_GT = 27  
 SEPARATOR\_GTE = 28  
 SEPARATOR\_LEFT\_BRACKET = 29  
 SEPARATOR\_LEFT\_FIGURE\_BRACKET = 30  
 SEPARATOR\_LT = 31  
 SEPARATOR\_LTE = 32  
 SEPARATOR\_MINUS = 33  
 SEPARATOR\_MULTIPLICATION = 34  
 SEPARATOR\_NOT = 35  
 SEPARATOR\_NOT\_EQUALS = 36  
 SEPARATOR\_OR = 37  
 SEPARATOR\_PLUS = 38  
 SEPARATOR\_RIGHT\_BRACKET = 39  
 SEPARATOR\_RIGHT\_FIGURE\_BRACKET = 40  
 SEPARATOR\_SEMICOLON = 41  
 SEPARATOR\_COMMA = 42  
  
 UNRESOLVED = 0  
 EOF = -1  
  
  
KEYWORDS = {  
 'begin': Lex.KEYWORD\_BEGIN,  
 'bool': Lex.KEYWORD\_BOOL,  
 'else': Lex.KEYWORD\_ELSE,  
 'end': Lex.KEYWORD\_END,  
 'false': Lex.KEYWORD\_FALSE,  
 'float': Lex.KEYWORD\_FLOAT,  
 'for': Lex.KEYWORD\_FOR,  
 'if': Lex.KEYWORD\_IF,  
 'int': Lex.KEYWORD\_INT,  
 'next': Lex.KEYWORD\_NEXT,  
 'readln': Lex.KEYWORD\_READLN,  
 'step': Lex.KEYWORD\_STEP,  
 'to': Lex.KEYWORD\_TO,  
 'true': Lex.KEYWORD\_TRUE,  
 'while': Lex.KEYWORD\_WHILE,  
 'writeln': Lex.KEYWORD\_WRITELN,  
}  
  
semantic = {  
 Lex.KEYWORD\_BEGIN: "begin",  
 Lex.KEYWORD\_BOOL: "bool",  
 Lex.KEYWORD\_ELSE: "else",  
 Lex.KEYWORD\_END: "end",  
 Lex.KEYWORD\_FALSE: "false",  
 Lex.KEYWORD\_FLOAT: "float",  
 Lex.KEYWORD\_FOR: "for",  
 Lex.KEYWORD\_IF: "if",  
 Lex.KEYWORD\_INT: "int",  
 Lex.KEYWORD\_NEXT: "next",  
 Lex.KEYWORD\_READLN: "readln",  
 Lex.KEYWORD\_STEP: "step",  
 Lex.KEYWORD\_TO: "to",  
 Lex.KEYWORD\_TRUE: "true",  
 Lex.KEYWORD\_WHILE: "while",  
 Lex.KEYWORD\_WRITELN: "writeln",  
  
 Lex.IDENTIFIER: "идентификатор",  
  
 Lex.NUMBER\_BIN: "двоичное число",  
 Lex.NUMBER\_OCT: "восьмиричное число",  
 Lex.NUMBER\_DEC: "десятичное число",  
 Lex.NUMBER\_HEX: "шестнадцатиричное число",  
 Lex.NUMBER\_FRACTIONAL: "дробное число",  
  
 Lex.SEPARATOR\_AND: "and",  
 Lex.SEPARATOR\_ASSIGNMENT: ":=",  
 Lex.SEPARATOR\_DIVISION: "/",  
 Lex.SEPARATOR\_EQUALS: "==",  
 Lex.SEPARATOR\_GT: ">",  
 Lex.SEPARATOR\_GTE: ">=",  
 Lex.SEPARATOR\_LEFT\_BRACKET: "(",  
 Lex.SEPARATOR\_LEFT\_FIGURE\_BRACKET: "{",  
 Lex.SEPARATOR\_LT: "<",  
 Lex.SEPARATOR\_LTE: "<=",  
 Lex.SEPARATOR\_MINUS: "-",  
 Lex.SEPARATOR\_MULTIPLICATION: "\*",  
 Lex.SEPARATOR\_NOT: "!",  
 Lex.SEPARATOR\_NOT\_EQUALS: "!=",  
 Lex.SEPARATOR\_OR: "||",  
 Lex.SEPARATOR\_PLUS: "+",  
 Lex.SEPARATOR\_RIGHT\_BRACKET: ")",  
 Lex.SEPARATOR\_RIGHT\_FIGURE\_BRACKET: "}",  
 Lex.SEPARATOR\_SEMICOLON: ";",  
 Lex.SEPARATOR\_COMMA: ",",  
 Lex.EOF: "Конец файла"  
}  
  
  
@dataclass  
class Lexeme:  
 lex: Lex  
 value: str  
 line: int  
 symbol: int  
 error: str

Листинг 11.5 – src/\_\_init\_\_.py

\_\_version\_\_ = "0.0.1"  
  
  
from src.cpp\_translator import translate as cpp\_translate  
from src.interpreter import run  
from src.lexer import Lexer  
from src.parser import Parser  
from src.python\_translator import translate as py\_translate  
from src.text\_driver import get\_filename, setup\_source  
from src.tree import optimize\_tree

Листинг 11.6 – src/tree/\_\_init\_\_.py

from .optimizations import \*  
from .tree import \*

Листинг 11.7 – src/tree/optimizations.py

from src.lang import Integer  
from src.tree.tree import \*  
  
  
def division\_to\_mod\_check(ast: ASTTyped) -> Union[ASTBinOperation | None]:  
 if ast.a\_type == ASTType.BIN\_OP and ast.operation == BinOperations.eq:  
 if (ast.right.t\_type == Types.int and  
 ast.left.t\_type == Types.int and  
 ast.right.a\_type == ASTType.VAR and  
 ast.left.a\_type == ASTType.BIN\_OP and  
 ast.left.operation == BinOperations.mul):  
 node: ASTBinOperation = ast.left  
 first\_var\_name = ast.right.name  
 if (  
 node.right.a\_type == ASTType.VAR and node.left.a\_type == ASTType.BIN\_OP and node.left.operation == BinOperations.div):  
 second\_var\_name = node.right.name  
 node: ASTBinOperation = node.left  
 if (node.left.a\_type == ASTType.VAR and node.right.a\_type == ASTType.VAR):  
 if node.left.name == first\_var\_name and node.right.name == second\_var\_name:  
 return ASTBinOperation(ASTBinOperation(ast.right, ast.left.right, BinOperations.mod),  
 ASTConst(Types.int, Integer(0)), BinOperations.eq)  
  
  
def optimize\_tree(ast: ASTNode):  
 while ast is not None:  
 optimize\_one(ast)  
 ast = ast.next\_node  
  
  
def optimize\_expression(ast: ASTTyped) -> Union[ASTNode | None]:  
 mod\_check\_operation = division\_to\_mod\_check(ast)  
 if mod\_check\_operation is not None:  
 return mod\_check\_operation  
  
  
def optimize\_one(ast: ASTNode):  
 if ast.a\_type == ASTType.ASSIGNMENT:  
 ast: ASTAssignment  
 r = optimize\_expression(ast.value)  
 if r is not None:  
 ast.value = r  
  
 elif ast.a\_type == ASTType.IF:  
 ast: ASTIf  
 r = optimize\_expression(ast.condition)  
 if r is not None:  
 ast.condition = r  
 node = ast.branch  
 while node is not None:  
 optimize\_one(node)  
 node = node.next\_node  
 if ast.else\_branch is not None:  
 node = ast.else\_branch  
 while node is not None:  
 optimize\_one(node)  
 node = node.next\_node  
  
 elif ast.a\_type == ASTType.Loop:  
 ast: ASTLoop  
 r = optimize\_expression(ast.condition)  
 if r is not None:  
 ast.condition = r  
  
 node = ast.body  
 while node is not None:  
 optimize\_one(node)  
 node = node.next\_node  
  
 elif ast.a\_type == ASTType.ForLoop:  
 ast: ASTForLoop  
 r = optimize\_expression(ast.step)  
 if r is not None:  
 ast.step = r  
  
 r = optimize\_expression(ast.assignment.value)  
 if r is not None:  
 ast.assignment.value = r  
  
 r = optimize\_expression(ast.condition.right)  
 if r is not None:  
 ast.condition.right = r  
  
 node = ast.body  
 while node is not None:  
 optimize\_one(node)  
 node = node.next\_node  
  
 elif ast.a\_type == ASTType.IN:  
 pass  
  
 elif ast.a\_type == ASTType.OUT:  
 ast: ASTOut  
 r = optimize\_expression(ast.expressions)  
 if r is not None:  
 ast.expressions = r  
  
 prev = ast.expressions  
 node = ast.expressions.next\_node  
  
 while node is not None:  
 r = optimize\_expression(node)  
 if r is not None:  
 prev.next\_node = r  
  
 prev = node  
 node = node.next\_node

Листинг 11.8 – src/tree/tree.py

from enum import Enum  
from typing import Any, Union  
  
from src.lang import BinOperations, RELATION\_OPERATORS, Types  
  
  
class ASTType(Enum):  
 ASSIGNMENT = 0  
 IF = 1  
 Loop = 2  
 IN = 3  
 OUT = 4  
 DECL = 5  
 U\_OP = 6  
 BIN\_OP = 7  
 CONST = 8  
 VAR = 9  
 ForLoop = 10  
  
  
class ASTNode:  
 """  
 Базовый узел, имеет тип узла, ссылку на следующий узел  
 """  
 a\_type: ASTType  
 next\_node: Any # ASTNode  
  
 def \_\_init\_\_(self, a\_type: ASTType):  
 self.a\_type = a\_type  
 self.next\_node = None  
  
  
class ASTTyped(ASTNode):  
 """  
 Типизированный узел, может быть переменной, бинарной или унарной операцией, константой  
 """  
 t\_type: Types  
  
 def \_\_init\_\_(self, t\_type: Types, a\_type: ASTType):  
 super().\_\_init\_\_(a\_type)  
 self.t\_type = t\_type  
  
  
class ASTConst(ASTTyped):  
 """  
 Константа  
 """  
 value: Any  
  
 def \_\_init\_\_(self, t\_type: Types, value: Any):  
 super().\_\_init\_\_(t\_type, ASTType.CONST)  
 self.value = value  
  
  
class ASTVar(ASTTyped):  
 """  
 Переменная  
 """  
 name: str  
  
 def \_\_init\_\_(self, t\_type: Types, name: str):  
 super().\_\_init\_\_(t\_type, ASTType.VAR)  
 self.name = name  
  
  
class ASTAssignment(ASTNode):  
 """  
 Оператор присваивания  
 """  
 var: ASTVar  
 value: ASTTyped  
  
 def \_\_init\_\_(self, var: ASTVar, value: ASTTyped):  
 super().\_\_init\_\_(ASTType.ASSIGNMENT)  
 self.var = var  
 self.value = value  
  
  
class ASTIf(ASTNode):  
 """  
 Условный оператор  
 """  
 condition: ASTTyped  
 branch: ASTNode # Список  
 else\_branch: Union[ASTNode, None] # Список  
  
 def \_\_init\_\_(self, condition: ASTTyped, branch: ASTNode, else\_branch: ASTNode = None):  
 super().\_\_init\_\_(ASTType.IF)  
 self.condition = condition  
 self.branch = branch  
 self.else\_branch = else\_branch  
  
  
class ASTBinOperation(ASTTyped):  
 """  
 Бинарная операция  
 """  
 left: ASTTyped  
 right: ASTTyped  
 operation: BinOperations  
  
 def \_\_init\_\_(self, left: ASTTyped, right: ASTTyped, operation: BinOperations):  
 if operation in RELATION\_OPERATORS:  
 super().\_\_init\_\_(Types.bool, ASTType.BIN\_OP)  
 else:  
 super().\_\_init\_\_(left.t\_type, ASTType.BIN\_OP)  
 self.left = left  
 self.right = right  
 self.operation = operation  
  
  
class ASTUOperation(ASTTyped):  
 """  
 Унарная операция (только '!' - отрицание)  
 """  
 operand = ASTTyped  
  
 def \_\_init\_\_(self, operand: ASTTyped, parent=None):  
 super().\_\_init\_\_(operand.t\_type, ASTType.U\_OP)  
 self.operand = operand  
  
  
class ASTLoop(ASTNode):  
 """  
 Цикл  
 """  
 condition: ASTTyped  
 body: ASTNode  
  
 def \_\_init\_\_(self, condition: ASTTyped, body: ASTNode):  
 super().\_\_init\_\_(ASTType.Loop)  
 self.condition = condition  
 self.body = body  
  
  
class ASTForLoop(ASTNode):  
 """  
 Цикл for  
 step - подразумевается, что вычисляется один раз и превращается в константу  
 """  
 assignment: ASTAssignment  
 condition: ASTBinOperation  
 body: ASTNode  
 step: ASTTyped  
  
 def \_\_init\_\_(self, assignment: ASTAssignment, expression: ASTTyped, body: ASTNode, step: ASTTyped):  
 super().\_\_init\_\_(ASTType.ForLoop)  
 self.assignment = assignment  
 self.condition = ASTBinOperation(assignment.var, expression, BinOperations.lt)  
 self.body = body  
 self.step = step  
  
  
class ASTDeclaration(ASTTyped):  
 """  
 Оператор определения переменных  
 """  
 variables: ASTVar # список  
  
 def \_\_init\_\_(self, variables: ASTVar, parent=None):  
 super().\_\_init\_\_(variables.t\_type, ASTType.DECL)  
 self.variables = variables  
  
  
class ASTIn(ASTNode):  
 """  
 Оператор ввода  
 """  
 variables: ASTVar # список  
  
 def \_\_init\_\_(self, variables: ASTVar):  
 super().\_\_init\_\_(ASTType.IN)  
 self.variables = variables  
  
  
class ASTOut(ASTNode):  
 """  
 Оператор вывода  
 """  
 expressions: ASTTyped # список  
  
 def \_\_init\_\_(self, expressions: ASTTyped):  
 super().\_\_init\_\_(ASTType.OUT)  
 self.expressions = expressions

Листинг 11.9 – src/python\_translator/\_\_init\_\_.py

from .translator import translate

Листинг 11.10 – src/python\_translator/translator.py

from src.tree import \*  
  
type\_enum\_to\_type\_str\_table = {  
 Types.int: "int",  
 Types.float: "float",  
 Types.bool: "bool"  
}  
  
  
def translate(ast: ASTNode):  
 r = "\n\n"  
 r += "def main():\n"  
 while ast is not None:  
 r += translate\_operator(ast)  
 ast = ast.next\_node  
 r += "\n\n"  
 r += "if \_\_name\_\_ == '\_\_main\_\_':\n\tmain()\n"  
 return r  
  
  
def translate\_expression(ast: ASTTyped) -> str:  
 mod\_check\_operation = division\_to\_mod\_check(ast)  
 if mod\_check\_operation is not None:  
 return translate\_expression(mod\_check\_operation)  
  
 if ast.a\_type == ASTType.CONST:  
 ast: ASTConst  
 return str(ast.value)  
 elif ast.a\_type == ASTType.VAR:  
 ast: ASTVar  
 return str(ast.name)  
  
 elif ast.a\_type == ASTType.U\_OP:  
 ast: ASTUOperation  
 return "(not " + translate\_expression(ast.operand) + ")"  
 elif ast.a\_type == ASTType.BIN\_OP:  
 ast: ASTBinOperation  
 if ast.operation == BinOperations.sum:  
 return "(" + translate\_expression(ast.left) + " + " + translate\_expression(ast.right) + ")"  
 elif ast.operation == BinOperations.diff:  
 return "(" + translate\_expression(ast.left) + " - " + translate\_expression(ast.right) + ")"  
 elif ast.operation == BinOperations.mul:  
 return "(" + translate\_expression(ast.left) + " \* " + translate\_expression(ast.right) + ")"  
 elif ast.operation == BinOperations.div:  
 if ast.t\_type == Types.int:  
 return "(" + translate\_expression(ast.left) + " // " + translate\_expression(ast.right) + ")"  
 else:  
 return "(" + translate\_expression(ast.left) + " / " + translate\_expression(ast.right) + ")"  
 elif ast.operation == BinOperations.alt:  
 return "(" + translate\_expression(ast.left) + " or " + translate\_expression(ast.right) + ")"  
 elif ast.operation == BinOperations.con:  
 return "(" + translate\_expression(ast.left) + " and " + translate\_expression(ast.right) + ")"  
 elif ast.operation == BinOperations.lt:  
 return "(" + translate\_expression(ast.left) + " < " + translate\_expression(ast.right) + ")"  
 elif ast.operation == BinOperations.lte:  
 return "(" + translate\_expression(ast.left) + " <= " + translate\_expression(ast.right) + ")"  
 elif ast.operation == BinOperations.gt:  
 return "(" + translate\_expression(ast.left) + " > " + translate\_expression(ast.right) + ")"  
 elif ast.operation == BinOperations.gte:  
 return "(" + translate\_expression(ast.left) + " >= " + translate\_expression(ast.right) + ")"  
 elif ast.operation == BinOperations.eq:  
 return "(" + translate\_expression(ast.left) + " == " + translate\_expression(ast.right) + ")"  
 elif ast.operation == BinOperations.neq:  
 return "(" + translate\_expression(ast.left) + " != " + translate\_expression(ast.right) + ")"  
 elif ast.operation == BinOperations.mod:  
 return "(" + translate\_expression(ast.left) + " % " + translate\_expression(ast.right) + ")"  
 else:  
 raise ValueError("Неопределённая операция")  
  
  
def translate\_operator(ast: ASTNode, tabs: int = 1):  
 if ast.a\_type == ASTType.DECL:  
 ast: ASTDeclaration  
 node = ast.variables  
 r = ""  
 while node is not None:  
 r += "\t" \* tabs + node.name  
 r += ": " + type\_enum\_to\_type\_str\_table.get(ast.t\_type) + "\n"  
 node = node.next\_node  
  
 elif ast.a\_type == ASTType.ASSIGNMENT:  
 ast: ASTAssignment  
 r = "\t" \* tabs + ast.var.name + " = " + translate\_expression(ast.value)  
 r += "\n"  
  
 elif ast.a\_type == ASTType.IF:  
 ast: ASTIf  
 condition = translate\_expression(ast.condition)  
 r = "\n" + "\t" \* tabs + "if " + condition + ":" + "\n"  
 node = ast.branch  
 while node is not None:  
 r += translate\_operator(node, tabs + 1)  
 node = node.next\_node  
 if ast.else\_branch is not None:  
 r += "\t" \* tabs + "else:\n"  
 node = ast.else\_branch  
 while node is not None:  
 r += translate\_operator(node, tabs + 1)  
 node = node.next\_node  
  
 elif ast.a\_type == ASTType.Loop:  
 ast: ASTLoop  
 condition = translate\_expression(ast.condition)  
 r = "\n" + "\t" \* tabs + "while " + condition + ":" + "\n"  
 node = ast.body  
 while node is not None:  
 r += translate\_operator(node, tabs + 1)  
 node = node.next\_node  
  
 elif ast.a\_type == ASTType.ForLoop:  
 ast: ASTForLoop  
 step = translate\_expression(ast.step)  
  
 start = translate\_expression(ast.assignment.value)  
 end = translate\_expression(ast.condition.right)  
  
 r = ("\n" + "\t" \* tabs + "for " + ast.assignment.var.name +  
 " in range(" + start + ", " + end + ", " + step + "):\n")  
 node = ast.body  
 while node is not None:  
 r += translate\_operator(node, tabs + 1)  
 node = node.next\_node  
  
 elif ast.a\_type == ASTType.IN:  
 ast: ASTIn  
 node = ast.variables  
 r = ""  
 while node is not None:  
 r += "\t" \* tabs + node.name + " = "  
 if node.t\_type == Types.int:  
 r += "int("  
 elif node.t\_type == Types.float:  
 r += "float("  
 else:  
 r += "bool("  
 r += "input())\n"  
 node = node.next\_node  
  
 elif ast.a\_type == ASTType.OUT:  
 ast: ASTOut  
 node = ast.expressions  
 r = ""  
 while node is not None:  
 r += "\t" \* tabs + "print(" + translate\_expression(node) + ")\n"  
 node = node.next\_node  
 else:  
 raise ValueError("Оператор не определён")  
  
 return r

Листинг 11.11 – src/parser/\_\_init\_\_.py

from .parser import Parser

Листинг 11.12 – src/parser/name\_table.py

from typing import Any, Dict, List  
  
from src.constants import Lexeme  
from src.errors import ctx\_error  
from src.lang.lang\_base\_types import Types  
  
  
class TableItem:  
 name: str  
 t\_type: Types  
 value: Any  
 is\_assigned: bool  
 readonly: bool  
  
 def \_\_init\_\_(self, name: str, t\_type: Types, value: Any, readonly=False):  
 self.name = name  
 self.t\_type = t\_type  
 self.value = value  
 self.is\_assigned = False  
 self.readonly = readonly  
  
  
tables: List[Dict[str, TableItem]] = []  
  
  
def add\_name(item: TableItem):  
 # без проверок  
 tables[-1][item.name] = item  
  
  
def find\_name(key: str) -> TableItem:  
 for table in tables[::-1]:  
 r = table.get(key)  
 if r is not None:  
 return r  
 return None  
  
  
def new\_name(t\_type: Types, identifier: Lexeme, readonly=False):  
 exist = find\_name(identifier.value)  
 if exist is not None:  
 ctx\_error("Повторное определение имени", identifier)  
  
 add\_name(TableItem(  
 identifier.value,  
 t\_type,  
 None,  
 readonly=readonly  
 ))  
  
  
def open\_scope():  
 tables.append({})  
  
  
def close\_scope():  
 tables.pop(-1)

Листинг 11.13 – src/parser/parser.py

from typing import Generator, Tuple  
  
from src.constants import Lex, Lexeme  
from src.errors import ctx\_error, expected, expected\_msg  
from src.lang import get\_type\_number\_from\_lex, Types  
from src.lang.lang\_base\_types import Boolean, Float, get\_bin\_operation\_from\_lex, Integer  
from src.lexer import Lexer  
from src.parser.name\_table import close\_scope, find\_name, new\_name, open\_scope  
from src.tree import ASTAssignment, ASTBinOperation, ASTConst, ASTDeclaration, ASTForLoop, ASTIf, ASTIn, ASTLoop, \  
 ASTNode, ASTOut, \  
 ASTTyped, ASTUOperation, ASTVar, optimize\_tree  
  
  
class Parser:  
 lexer: Generator  
 lexeme: Lexeme  
  
 def \_\_init\_\_(self, lexer: Lexer):  
 self.lexer = iter(lexer.get\_lex())  
 self.lexeme = self.\_\_new\_lex()  
  
 def \_\_new\_lex(self):  
 try:  
 self.lexeme = next(self.lexer)  
 except StopIteration:  
 pass  
 else:  
 return self.lexeme  
  
 def \_\_get\_lex(self):  
 return self.lexeme  
  
 def \_\_skip\_lex(self, lex: Lex):  
 if self.lexeme.lex == lex:  
 self.\_\_new\_lex()  
 else:  
 expected(lex, self.lexeme)  
  
 def parse(self) -> ASTNode:  
 open\_scope()  
 ast = self.\_\_program\_parser()  
 close\_scope()  
 optimize\_tree(ast)  
 return ast  
  
 def \_\_identifier\_declaration(self, t\_type: Types, identifier: Lexeme, readonly=False):  
 new\_name(t\_type, identifier, readonly=readonly)  
  
 def \_\_description\_parser(self) -> ASTNode:  
 if self.lexeme.lex not in (Lex.KEYWORD\_INT, Lex.KEYWORD\_FLOAT, Lex.KEYWORD\_BOOL):  
 expected\_msg("Тип", self.lexeme)  
  
 identifier\_type = get\_type\_number\_from\_lex(self.lexeme.lex)  
 self.\_\_new\_lex()  
 if self.lexeme.lex != Lex.IDENTIFIER:  
 expected(Lex.IDENTIFIER, self.lexeme)  
  
 identifier = self.lexeme  
 self.\_\_identifier\_declaration(identifier\_type, identifier)  
 self.\_\_new\_lex()  
  
 head = ASTVar(identifier\_type, identifier.value)  
 tail = head  
  
 while self.lexeme.lex == Lex.SEPARATOR\_COMMA:  
 self.\_\_new\_lex()  
 if self.lexeme.lex != Lex.IDENTIFIER:  
 expected(Lex.IDENTIFIER, self.lexeme)  
  
 self.\_\_identifier\_declaration(identifier\_type, self.lexeme)  
 tail.next\_node = ASTVar(identifier\_type, self.lexeme.value)  
 tail = tail.next\_node  
 self.\_\_new\_lex()  
  
 return ASTDeclaration(head)  
  
 def \_\_operator\_combine\_parser(self) -> Tuple[ASTNode, ASTNode]:  
 self.\_\_skip\_lex(Lex.KEYWORD\_BEGIN)  
 h\_head, h\_tail = self.\_\_operator\_parser()  
 tail = h\_tail  
 while self.lexeme.lex == Lex.SEPARATOR\_SEMICOLON:  
 self.\_\_new\_lex()  
  
 t\_head, t\_tail = self.\_\_operator\_parser()  
 tail.next\_node = t\_head  
  
 tail = t\_tail  
  
 self.\_\_skip\_lex(Lex.KEYWORD\_END)  
 return h\_head, tail  
  
 def \_\_identifier\_assignment(self, identifier: Lexeme):  
 find\_name(identifier.value).is\_assigned = True  
  
 def \_\_factor\_parser(self) -> ASTTyped:  
 if self.lexeme.lex in (Lex.IDENTIFIER, Lex.NUMBER\_BIN, Lex.NUMBER\_OCT,  
 Lex.NUMBER\_DEC, Lex.NUMBER\_HEX, Lex.NUMBER\_FRACTIONAL,  
 Lex.KEYWORD\_TRUE, Lex.KEYWORD\_FALSE, Lex.SEPARATOR\_NOT,  
 Lex.SEPARATOR\_LEFT\_BRACKET):  
 if self.lexeme.lex == Lex.SEPARATOR\_NOT:  
 self.\_\_new\_lex()  
 lex\_exp\_start = self.lexeme  
 node = self.\_\_factor\_parser()  
 if node.t\_type != Types.bool:  
 ctx\_error("Унарная операция поддерживается только для типа bool", lex\_exp\_start)  
  
 node = ASTUOperation(node)  
  
 return node  
  
 elif self.lexeme.lex == Lex.SEPARATOR\_LEFT\_BRACKET:  
 self.\_\_new\_lex()  
 node = self.\_\_expression\_parser()  
 self.\_\_skip\_lex(Lex.SEPARATOR\_RIGHT\_BRACKET)  
 return node  
 else:  
 factor = self.lexeme  
 self.\_\_new\_lex()  
 if factor.lex in (Lex.KEYWORD\_TRUE, Lex.KEYWORD\_FALSE):  
 return ASTConst(Types.bool, Boolean.from\_string(factor.value))  
 elif factor.lex == Lex.NUMBER\_FRACTIONAL:  
 return ASTConst(Types.float, Float.from\_string(factor.value))  
 elif factor.lex == Lex.IDENTIFIER:  
 t\_item = find\_name(factor.value)  
 if t\_item is None:  
 ctx\_error("Переменная не объявлена в этой области видимости", factor)  
 elif not t\_item.is\_assigned:  
 ctx\_error("Переменная не инициализированна", factor)  
  
 return ASTVar(t\_item.t\_type, factor.value)  
 return ASTConst(Types.int, Integer.from\_string(factor.value))  
 else:  
 expected\_msg("Выражение", self.lexeme)  
  
 def \_\_multiplication\_parser(self) -> ASTTyped:  
 left\_node = self.\_\_factor\_parser()  
 node = left\_node  
 while self.lexeme.lex in (Lex.SEPARATOR\_MULTIPLICATION, Lex.SEPARATOR\_DIVISION, Lex.SEPARATOR\_AND):  
 operation = self.lexeme  
 self.\_\_new\_lex()  
 right\_node = self.\_\_factor\_parser()  
 if ((left\_node.t\_type == Types.bool or right\_node.t\_type == Types.bool) and  
 operation.lex != Lex.SEPARATOR\_AND):  
 ctx\_error("Неподдерживаемая операция для типа bool, возможно вы имели ввиду '&&'", operation)  
 elif left\_node.t\_type != right\_node.t\_type:  
 ctx\_error("Типы операндов не совпадают", operation)  
 elif left\_node.t\_type in (Types.int, Types.float) and operation.lex == Lex.SEPARATOR\_AND:  
 ctx\_error("Неподдерживаемая операция для числового типа", operation)  
  
 node = ASTBinOperation(node, right\_node, get\_bin\_operation\_from\_lex(operation.lex))  
 return node  
  
 def \_\_summa\_parser(self) -> ASTTyped:  
 left\_node = self.\_\_multiplication\_parser()  
 node = left\_node  
 while self.lexeme.lex in (Lex.SEPARATOR\_PLUS, Lex.SEPARATOR\_MINUS, Lex.SEPARATOR\_OR):  
 operation = self.lexeme  
 self.\_\_new\_lex()  
 right\_node = self.\_\_multiplication\_parser()  
 if ((left\_node.t\_type == Types.bool or right\_node.t\_type == Types.bool) and  
 operation.lex != Lex.SEPARATOR\_OR):  
 ctx\_error("Неподдерживаемая операция для типа bool, возможно вы имели ввиду '||'", operation)  
 elif left\_node.t\_type != right\_node.t\_type:  
 ctx\_error("Типы операндов не совпадают", operation)  
 elif left\_node.t\_type in (Types.int, Types.float) and operation.lex == Lex.SEPARATOR\_OR:  
 ctx\_error("Неподдерживаемая операция для числового типа", operation)  
 node = ASTBinOperation(node, right\_node, get\_bin\_operation\_from\_lex(operation.lex))  
 return node  
  
 def \_\_expression\_parser(self) -> ASTTyped:  
 left\_node = self.\_\_summa\_parser()  
 node = left\_node  
 while self.lexeme.lex in (Lex.SEPARATOR\_EQUALS, Lex.SEPARATOR\_NOT\_EQUALS, Lex.SEPARATOR\_LT,  
 Lex.SEPARATOR\_LTE, Lex.SEPARATOR\_GT, Lex.SEPARATOR\_GTE):  
 operation = self.lexeme  
 self.\_\_new\_lex()  
 right\_node = self.\_\_summa\_parser()  
 if (left\_node.t\_type == Types.bool or right\_node.t\_type == Types.bool) and operation.lex not in (  
 Lex.SEPARATOR\_EQUALS, Lex.SEPARATOR\_NOT\_EQUALS):  
 ctx\_error("Неподдерживаемая операция для типа bool", operation)  
 elif left\_node.t\_type != right\_node.t\_type:  
 ctx\_error("Типы операндов не совпадают", operation)  
 node = ASTBinOperation(node, right\_node, get\_bin\_operation\_from\_lex(operation.lex))  
 return node  
  
 def \_\_operator\_assignment\_parser(self) -> ASTAssignment:  
 if self.lexeme.lex != Lex.IDENTIFIER:  
 expected(Lex.IDENTIFIER, self.lexeme)  
  
 identifier = self.lexeme  
 self.\_\_new\_lex()  
 self.\_\_skip\_lex(Lex.SEPARATOR\_ASSIGNMENT)  
 exp\_start\_lex = self.lexeme  
 node\_expression = self.\_\_expression\_parser()  
 identifier\_type = find\_name(identifier.value)  
 if identifier\_type is None:  
 ctx\_error("Переменная не объявлена", identifier)  
 elif identifier\_type.readonly:  
 ctx\_error(f"Переменная доступна только для чтения", identifier)  
 elif identifier\_type.t\_type != node\_expression.t\_type:  
 ctx\_error(f"Присваиваемое выражение имеет отличный тип от типа переменной, \  
ожидается {identifier\_type.t\_type.name}", exp\_start\_lex)  
 var = ASTVar(identifier\_type.t\_type, identifier\_type.name)  
 self.\_\_identifier\_assignment(identifier)  
  
 return ASTAssignment(var, node\_expression)  
  
 def \_\_operator\_if\_parser(self) -> ASTIf:  
 self.\_\_skip\_lex(Lex.KEYWORD\_IF)  
 self.\_\_skip\_lex(Lex.SEPARATOR\_LEFT\_BRACKET)  
 lex\_condition\_start = self.lexeme  
 node\_condition = self.\_\_expression\_parser()  
 if node\_condition.t\_type != Types.bool:  
 ctx\_error("Условие должно быть типа bool", lex\_condition\_start)  
 self.\_\_skip\_lex(Lex.SEPARATOR\_RIGHT\_BRACKET)  
 node\_branch, \_ = self.\_\_operator\_parser()  
  
 if self.lexeme.lex == Lex.KEYWORD\_ELSE:  
 self.\_\_new\_lex()  
 node\_else\_branch, \_ = self.\_\_operator\_parser()  
  
 return ASTIf(node\_condition, node\_branch, else\_branch=node\_else\_branch)  
  
 return ASTIf(node\_condition, node\_branch)  
  
 def \_\_operator\_for\_parser(self) -> ASTForLoop:  
 self.\_\_skip\_lex(Lex.KEYWORD\_FOR)  
 lex\_assignment\_start = self.lexeme  
  
 open\_scope()  
 identifier\_type = Types.int  
 self.\_\_identifier\_declaration(identifier\_type, lex\_assignment\_start)  
  
 node\_assignment = self.\_\_operator\_assignment\_parser()  
 if node\_assignment.var.t\_type != Types.int:  
 ctx\_error("Переменная цикла for должна быть типа int, иначе используйте while", lex\_assignment\_start)  
  
 identifier = find\_name(lex\_assignment\_start.value)  
 identifier.readonly = True  
  
 self.\_\_skip\_lex(Lex.KEYWORD\_TO)  
 lex\_expression\_start = self.lexeme  
 node\_expression = self.\_\_expression\_parser()  
 if node\_expression.t\_type != Types.int:  
 ctx\_error("Условие цикла for должно быть типа int", lex\_expression\_start)  
  
 if self.lexeme.lex == Lex.KEYWORD\_STEP:  
 self.\_\_new\_lex()  
 lex\_expression\_start = self.lexeme  
 node\_step = self.\_\_expression\_parser()  
  
 if node\_step.t\_type != Types.int:  
 ctx\_error(f"Несоответствие типов, выражение в step должно быть типа int \  
({node\_assignment.var.t\_type.name})", lex\_expression\_start)  
 else:  
 node\_step = ASTConst(Types.int, Integer(1))  
  
 node\_body\_start, \_ = self.\_\_operator\_parser()  
  
 self.\_\_skip\_lex(Lex.KEYWORD\_NEXT)  
  
 node\_loop = ASTForLoop(node\_assignment, node\_expression, node\_body\_start, node\_step)  
 close\_scope()  
 return node\_loop  
  
 def \_\_operator\_while\_parser(self) -> ASTLoop:  
 self.\_\_skip\_lex(Lex.KEYWORD\_WHILE)  
 self.\_\_skip\_lex(Lex.SEPARATOR\_LEFT\_BRACKET)  
 lex\_condition\_start = self.lexeme  
 node\_condition = self.\_\_expression\_parser()  
 if node\_condition.t\_type != Types.bool:  
 ctx\_error("Условие должно быть типа bool", lex\_condition\_start)  
 self.\_\_skip\_lex(Lex.SEPARATOR\_RIGHT\_BRACKET)  
 node\_body, \_ = self.\_\_operator\_parser()  
 return ASTLoop(node\_condition, node\_body)  
  
 def \_\_operator\_readln\_parser(self) -> ASTIn:  
 self.\_\_skip\_lex(Lex.KEYWORD\_READLN)  
 if self.lexeme.lex != Lex.IDENTIFIER:  
 expected(Lex.IDENTIFIER, self.lexeme)  
  
 identifier = self.lexeme  
 var = find\_name(identifier.value)  
 if var is None:  
 ctx\_error("Переменная не объявлена", identifier)  
 elif var.readonly:  
 ctx\_error(f"Переменная доступна только для чтения", identifier)  
 self.\_\_identifier\_assignment(identifier)  
 self.\_\_new\_lex()  
  
 head = ASTVar(var.t\_type, var.name)  
 node = head  
  
 while self.lexeme.lex == Lex.SEPARATOR\_COMMA:  
 self.\_\_new\_lex()  
 if self.lexeme.lex != Lex.IDENTIFIER:  
 expected(Lex.IDENTIFIER, self.lexeme)  
 identifier = self.lexeme  
 var = find\_name(identifier.value)  
 if var is None:  
 ctx\_error("Переменная не объявлена", identifier)  
 self.\_\_identifier\_assignment(identifier)  
 node.next\_node = ASTVar(var.t\_type, var.name)  
 node = node.next\_node  
 self.\_\_new\_lex()  
  
 return ASTIn(head)  
  
 def \_\_operator\_writeln\_parser(self) -> ASTOut:  
 self.\_\_skip\_lex(Lex.KEYWORD\_WRITELN)  
 head = self.\_\_expression\_parser()  
 node = head  
  
 while self.lexeme.lex == Lex.SEPARATOR\_COMMA:  
 self.\_\_new\_lex()  
 node.next\_node = self.\_\_expression\_parser()  
 node = node.next\_node  
 return ASTOut(head)  
  
 def \_\_operator\_parser(self) -> Tuple[ASTNode, ASTNode]:  
 node = None  
 if self.lexeme.lex == Lex.KEYWORD\_BEGIN:  
 return self.\_\_operator\_combine\_parser()  
 elif self.lexeme.lex == Lex.IDENTIFIER:  
 node = self.\_\_operator\_assignment\_parser()  
 elif self.lexeme.lex == Lex.KEYWORD\_IF:  
 node = self.\_\_operator\_if\_parser()  
 elif self.lexeme.lex == Lex.KEYWORD\_FOR:  
 node = self.\_\_operator\_for\_parser()  
 elif self.lexeme.lex == Lex.KEYWORD\_WHILE:  
 node = self.\_\_operator\_while\_parser()  
 elif self.lexeme.lex == Lex.KEYWORD\_READLN:  
 node = self.\_\_operator\_readln\_parser()  
 elif self.lexeme.lex == Lex.KEYWORD\_WRITELN:  
 node = self.\_\_operator\_writeln\_parser()  
 else:  
 expected\_msg("Оператор", self.lexeme)  
 return node, node  
  
 def \_\_description\_or\_operator\_seq\_parser(self) -> Tuple[ASTNode, ASTNode]:  
 head = None  
 tail = None  
 while self.lexeme.lex in (Lex.KEYWORD\_INT, Lex.KEYWORD\_FLOAT, Lex.KEYWORD\_BOOL,  
 Lex.KEYWORD\_BEGIN, Lex.IDENTIFIER, Lex.KEYWORD\_IF,  
 Lex.KEYWORD\_FOR, Lex.KEYWORD\_WHILE, Lex.KEYWORD\_READLN,  
 Lex.KEYWORD\_WRITELN):  
 h\_head, h\_tail = self.\_\_description\_or\_operator\_parser()  
 if head is None:  
 head = h\_head  
 tail = h\_tail  
 else:  
 tail.next\_node = h\_head  
 tail = h\_tail  
 self.\_\_skip\_lex(Lex.SEPARATOR\_SEMICOLON)  
 return head, tail  
  
 def \_\_description\_or\_operator\_parser(self) -> Tuple[ASTNode, ASTNode]:  
 if self.lexeme.lex in (Lex.KEYWORD\_INT, Lex.KEYWORD\_FLOAT, Lex.KEYWORD\_BOOL):  
 node = self.\_\_description\_parser()  
 return node, node  
 else:  
 return self.\_\_operator\_parser()  
  
 def \_\_program\_parser(self) -> ASTNode:  
 self.\_\_skip\_lex(Lex.SEPARATOR\_LEFT\_FIGURE\_BRACKET)  
  
 h\_head, h\_tail = self.\_\_description\_or\_operator\_parser()  
 self.\_\_skip\_lex(Lex.SEPARATOR\_SEMICOLON)  
  
 t\_head, t\_tail = self.\_\_description\_or\_operator\_seq\_parser()  
 if t\_head is not None:  
 h\_tail.next\_node = t\_head  
  
 self.\_\_skip\_lex(Lex.SEPARATOR\_RIGHT\_FIGURE\_BRACKET)  
 self.\_\_skip\_lex(Lex.EOF)  
  
 return h\_head

Листинг 11.14 – src/lexer/\_\_init\_\_.py

from .lexer import Lexer

Листинг 11.15 – src/lexer/lexer.py

from src.constants import BASE\_SEPARATORS, KEYWORDS, Lex, Lexeme, States  
from src.errors import lex\_error  
from src.lexer.handlers import HandlerFactory  
from src.text\_driver import get\_source  
  
"""  
Приблизительная оценка:  
  
Обработчики разделителей работают по следующему принципу  
каждый такой обработчик кушает свой первый символ и следующий символ  
[ - start handler  
 ] - identifier handler -> lex 'a'  
 [ - start handler  
 ] - not\_equals handler -> lex '!='  
 [ - start handler  
 ] - identifier handler -> lex 'b'  
a!=b\n  
  
[ - start handler  
 ] - identifier handler -> lex 'a'  
 [ - start handler  
 ] - not equals handler -> lex '!'  
 [ - start handler  
 ] - identifier handler -> lex 'b'  
a!b\n  
  
на месте \n должен быть любой разделитель из SEPARATORS  
  
Каждый раз, когда распознаётся лексема, вызывается метод unget (в методе give\_lex), так как обработчик получает символ,  
не подходящий под выражение лексемы, например идентификатор получил символ '!', тогда он выдаёт лексему идентификатора,   
переходное состояние START, мы в методе get\_lex понимаем, что '!' уже не относится к идентификатору, поэтому и вызывается unget  
чтобы новое состояние START разбирала '!', а не пропускала его  
  
Также Start обрабатывает одиночные разделители по такому же принципу, неважно, что мы точно знаем, что после '(', ')', ';' и тд.  
точно не будет ничего дельного и можно сразу отдавать лексему, в данной программе важен общий случай алгоритма для простоты кода  
таким образом после '(' распознается r'.' - любой символ и мы только тогда отдадим лексему, потом произойдёт unget и так по кругу.  
"""  
  
  
class Lexer():  
 # строка для разбора  
 \_source: str  
  
 # текущий номер символа  
 \_index: int  
  
 # текущее состояние  
 \_state: States  
  
 # класс, возвращающий нам обёртку нужного генератора в зависимости от состояния,  
 # следит за тем, чтобы после состояний START и ER генератор был пересоздан  
 \_factory: HandlerFactory  
  
 # буфер для лексемы  
 \_buffer: str  
  
 # номер разбираемой строки  
 \_line: int  
  
 # номер начального символа разбираемой лексемы  
 \_symbol: int  
  
 # сообщение об ошибке последней нераспознанной лексемы  
 \_error\_message: str  
  
 def \_\_init\_\_(self):  
 self.\_source = get\_source()  
 self.\_index = 0  
 self.\_state = States.START  
 self.\_factory = HandlerFactory()  
 self.\_buffer = ""  
 self.\_line = 1  
 self.\_symbol = 1  
 self.\_error\_message = ""  
  
 def \_get\_char(self):  
 """  
 Метод возвращает следующий символ строки  
 Если символов не осталось, выбрасывается StopIteration  
 """  
 if self.\_index >= len(self.\_source):  
 raise StopIteration  
 char = self.\_source[self.\_index]  
 self.\_index += 1  
 return char  
  
 def \_unget\_char(self):  
 """  
 Метод декрементирует указатель на символ строки  
 """  
 if self.\_index > 0:  
 self.\_index -= 1  
  
 def \_give\_lex(self, lex: Lex):  
 """  
 Метод возвращает кортеж из всех известных параметрах о лексеме  
 Все возвращаемые анализатором лексемы проходят через эту функцию  
 """  
 self.\_unget\_char()  
 if lex == Lex.UNRESOLVED:  
 lex\_error(Lexeme(lex, self.\_buffer, self.\_line, self.\_symbol, self.\_error\_message))  
 return Lexeme(lex, self.\_buffer, self.\_line, self.\_symbol, "")  
  
 def get\_lex(self):  
 """  
 Генератор  
 Возвращает очередную лексему через вызов give\_lex  
 """  
  
 while True:  
 try:  
 char = self.\_get\_char()  
 except StopIteration:  
 if self.\_state == States.SEPARATOR\_COMMENT:  
 self.\_error\_message = "Комментарий должен быть закрыт"  
 yield self.\_give\_lex(Lex.UNRESOLVED)  
 self.\_buffer = ""  
 self.\_symbol -= 1  
 yield self.\_give\_lex(Lex.EOF)  
 return  
  
 lex: Lex = None  
 # генератор, который будет заниматься разбором следующего символа, выбирается в зависимости от состояния  
 handler = self.\_factory.get\_handler(self.\_state)  
  
 # новое состояние, в которое нам следует перейти  
 new\_state = handler.send(char)  
  
 # если получили лексему и новое состояние  
 if isinstance(new\_state, tuple):  
 lex, new\_state = new\_state  
  
 # если получили сообщение об ошибке и состояние ошибки  
 if isinstance(new\_state, list):  
 self.\_error\_message, new\_state = new\_state  
  
 # если закончился комментарий  
 if new\_state == States.STATE\_NULL:  
 new\_state = States.START  
 self.\_buffer = ""  
 self.\_unget\_char()  
 self.\_state = new\_state  
 continue  
  
 # выдаём распознанную лексему  
 elif lex is not None:  
 if lex == Lex.IDENTIFIER:  
 keyword = KEYWORDS.get(self.\_buffer)  
 if keyword is None:  
 yield self.\_give\_lex(lex)  
 else:  
 yield self.\_give\_lex(keyword)  
 else:  
 yield self.\_give\_lex(lex)  
  
 # накопление буфера (в случае если lex не None накопление не нужно, так лексема уже была выдана во вне)  
 # если lex не None, в буфер попадёт `мусор`, который собьёт счётчик символа в строке (symbol)  
 if char not in BASE\_SEPARATORS and lex is None:  
 self.\_buffer += char  
  
 if lex is not None:  
 # если была распознана лексема  
  
 self.\_symbol += len(self.\_buffer)  
 self.\_buffer = ""  
  
 if self.\_state == States.SEPARATOR\_COMMENT and new\_state == States.SEPARATOR\_COMMENT:  
 self.\_symbol += 1  
 self.\_state = new\_state  
  
 # подсчёт номера линии и символа в строке  
 if char in BASE\_SEPARATORS:  
 if lex is None:  
 if char == "\n":  
 self.\_line += 1  
 self.\_symbol = 1  
 else:  
 self.\_symbol += 1  
  
 if new\_state == States.ER:  
 """  
 Особый случай, отлавливает собития вида  
 '123c\n' - в конце состояние будет NUMBERHEX, следующее - ER  
 при этом ER не успеет отработать, \n будет проглочено обработчиком NUMBERHEX  
 на месте NUMBERHEX может быть другое состояние  
 поэтому мы явно возвращаем ошибочную лексему и очищаем буфер для разбора новой строки  
 также позволяет разделять лексемы  
 чтобы строка  
 '123e 10\n' была распознана как ошибочная лексема и лексема десятичного числа  
 """  
 self.\_symbol += len(self.\_buffer)  
 yield self.\_give\_lex(Lex.UNRESOLVED)  
 self.\_buffer = ""  
 self.\_state = States.START  
  
  
class DebugLexer(Lexer):  
 """  
 Нужен исключительно для тестов  
 Вместо лексемы выдаёт кортеж  
 Вместо вызова ошибки отдаёт неопределённую лексему  
 """  
  
 def \_give\_lex(self, lex: Lex):  
 """  
 Метод возвращает кортеж из всех известных параметрах о лексеме  
 Все возвращаемые анализатором лексемы проходят через эту функцию  
 """  
 self.\_unget\_char()  
 if lex == Lex.UNRESOLVED:  
 return lex, self.\_buffer, self.\_line, self.\_symbol, self.\_error\_message  
 return lex, self.\_buffer, self.\_line, self.\_symbol, ""

Листинг 11.16 – src/lexer/handlers.py

from typing import TypeVar  
  
from src.constants import BASE\_SEPARATORS, Lex, SEPARATORS, States  
  
state\_handler = TypeVar("state\_handler")  
  
"""  
Далее описаны генераторы, обрабатывающие отдельные состояния  
state\_{состояние}\_handler  
  
состояние может быть простым, например identificator, start и тд.,  
так и сложным - state\_letter\_e\_handler, имеющим свои подсостояния  
беспокоиться за это не нужно, внутренние состояния меняются самостоятельно  
  
Правила:  
 если обработчик вернул States.START, значит закончен разбор соответствующей лексемы  
 (если обработчик состояния IDENTIFICATOR, то лексема IDENTIFICATOR)  
   
 если обработчик вернул состояние {STATE}END, значит закончен разбор лексемы {STATE}  
   
 иначе обработчик возвращает новое состояние, которое может быть и таким же  
   
   
 состояния используются через оболочку renewable класса HandlerFactory,  
 метод get\_handler которого сам оборачивает в renewable обработчик  
 этот обработчик позволяет каждый раз, когда генератор заканчивает свою работу  
 (возвращает состояние, отличное от того, которое он обрабатывает) подменять его на новый,  
 чтобы состояние сложных генераторов всегда было начальным  
   
 Отдельно см. класс HandlerFactory  
"""  
  
  
def state\_start\_handler():  
 """  
 Стартовое состояние не только предрешает судьбу следующего символа, оно является таким же состоянием и может отдавать разделители  
 Состояние, которое было отдано (не кортеж) получит уже следующий символ, а тот, что был обработан тут - сохранится в буфере  
 """  
 state = 0  
 while True:  
 char: str = yield  
  
 if state == 0:  
 if char.isalpha() and char.isascii():  
 yield States.IDENTIFIER  
 elif char in ("0", "1"):  
 yield States.NUMBER\_BIN  
 elif char in ("2", "3", "4", "5", "6", "7"):  
 yield States.NUMBER\_OCT  
 elif char in ("8", "9"):  
 yield States.NUMBER\_DEC  
 elif char == ".":  
 yield States.FRACTIONAL  
 elif char in BASE\_SEPARATORS:  
 yield States.START  
 elif char in ("(", ")", "+", "-", "\*", "{", "}", ";", ","):  
 if char == "(":  
 state = Lex.SEPARATOR\_LEFT\_BRACKET, States.START  
 elif char == ")":  
 state = Lex.SEPARATOR\_RIGHT\_BRACKET, States.START  
 elif char == "+":  
 state = Lex.SEPARATOR\_PLUS, States.START  
 elif char == "-":  
 state = Lex.SEPARATOR\_MINUS, States.START  
 elif char == "\*":  
 state = Lex.SEPARATOR\_MULTIPLICATION, States.START  
 elif char == "{":  
 state = Lex.SEPARATOR\_LEFT\_FIGURE\_BRACKET, States.START  
 elif char == "}":  
 state = Lex.SEPARATOR\_RIGHT\_FIGURE\_BRACKET, States.START  
 elif char == ";":  
 state = Lex.SEPARATOR\_SEMICOLON, States.START  
 elif char == ",":  
 state = Lex.SEPARATOR\_COMMA, States.START  
 yield States.START  
 elif char == "!":  
 yield States.SEPARATOR\_NOT  
 elif char == "=":  
 yield States.SEPARATOR\_EQUALS  
 elif char == "<":  
 yield States.SEPARATOR\_LT  
 elif char == ">":  
 yield States.SEPARATOR\_GT  
 elif char == "|":  
 yield States.SEPARATOR\_OR  
 elif char == "&":  
 yield States.SEPARATOR\_AND  
 elif char == ":":  
 yield States.SEPARATOR\_ASSIGNMENT  
 elif char == "/":  
 yield States.SEPARATOR\_COMMENT  
 else:  
 yield ["Не распознано начало лексемы", States.ER]  
 else:  
 yield state  
  
  
def state\_identificator\_handler():  
 while True:  
 char: str = yield  
  
 if char.isalpha() and char.isascii() or char.isdigit():  
 yield States.IDENTIFIER  
 elif char in SEPARATORS:  
 yield Lex.IDENTIFIER, States.START  
 else:  
 yield ["Идентификатор может состоять только из цифр и латинских букв в любом регистре", States.ER]  
  
  
def state\_separator\_equals\_handler():  
 # equals '=='  
 state = 0  
 while True:  
 char: str = yield  
  
 if state == 0:  
 if char == "=":  
 state = 1  
 yield States.SEPARATOR\_EQUALS  
 else:  
 yield ["Лексема не распознана, возможно вы имели ввиду '==' ?", States.ER]  
 else:  
 yield Lex.SEPARATOR\_EQUALS, States.START  
  
  
def state\_separator\_lt\_handler():  
 state = 0  
 """  
 state = 0: - получаем символ '=' или отдаём лексему SEPARATOR\_LT  
 state = 1 - отдаём лексему SEPARATPR\_LTE  
 """  
 while True:  
 char: str = yield  
  
 if state == 0:  
 if char == "=":  
 state = 1  
 yield States.SEPARATOR\_LT  
 else:  
 yield Lex.SEPARATOR\_LT, States.START  
 else:  
 yield Lex.SEPARATOR\_LTE, States.START  
  
  
def state\_separator\_gt\_handler():  
 state = 0  
 while True:  
 char: str = yield  
  
 if state == 0:  
 if char == "=":  
 state = 1  
 yield States.SEPARATOR\_GT  
 else:  
 yield Lex.SEPARATOR\_GT, States.START  
 else:  
 yield Lex.SEPARATOR\_GTE, States.START  
  
  
def state\_separator\_or\_handler():  
 state = 0  
 while True:  
 char: str = yield  
  
 if state == 0:  
 if char == "|":  
 state = 1  
 yield States.SEPARATOR\_OR  
 else:  
 yield ["Лексема не распознана, возможно вы имели ввиду '||' ?", States.ER]  
 else:  
 yield Lex.SEPARATOR\_OR, States.START  
  
  
def state\_separator\_and\_handler():  
 state = 0  
 while True:  
 char: str = yield  
  
 if state == 0:  
 if char == "&":  
 state = 1  
 yield States.SEPARATOR\_AND  
 else:  
 yield ["Лексема не распознана, возможно вы имели ввиду '&&' ?", States.ER]  
 else:  
 yield Lex.SEPARATOR\_AND, States.START  
  
  
def state\_separator\_assignment\_handler():  
 state = 0  
 while True:  
 char: str = yield  
  
 if state == 0:  
 if char == "=":  
 state = 1  
 yield States.SEPARATOR\_ASSIGNMENT  
 else:  
 yield ["Лексема не распознана, возможно вы имели ввиду ':=' ?", States.ER]  
 else:  
 yield Lex.SEPARATOR\_ASSIGNMENT, States.START  
  
  
def state\_separator\_comment\_handler():  
 state = 0  
 """  
 state = 0 - символ деления или начало комментария  
 state = 1 - тело комментария  
 state = 2 - конец комментария  
 """  
 while True:  
 char: str = yield  
  
 if state == 0:  
 if char == "\*":  
 state = 1  
 yield States.SEPARATOR\_COMMENT  
 else:  
 yield Lex.SEPARATOR\_DIVISION, States.START  
 elif state == 1:  
 if char == "\*":  
 state = 2  
 yield States.SEPARATOR\_COMMENT  
 elif state == 2:  
 if char == "/":  
 state = 3  
 elif char == "\*":  
 pass  
 else:  
 state = 1  
 yield States.SEPARATOR\_COMMENT  
 else:  
 yield States.STATE\_NULL  
  
  
def state\_separator\_not\_handler():  
 state = 0  
 """  
 state = 0: - получаем символ '=' или отдаём лексему SEPARATOR\_NOT  
 state = 1 - отдаём лексему SEPARATOR\_NOT\_EQUALS  
 """  
 while True:  
 char: str = yield  
  
 if state == 0:  
 if char == "=":  
 state = 1  
 yield States.SEPARATOR\_NOT  
 else:  
 yield Lex.SEPARATOR\_NOT, States.START  
 else:  
 yield Lex.SEPARATOR\_NOT\_EQUALS, States.START  
  
  
def state\_number\_bin\_handler():  
 while True:  
 char: str = yield  
  
 if char in ("0", "1"):  
 yield States.NUMBER\_BIN  
 elif char in ("2", "3", "4", "5", "6", "7"):  
 yield States.NUMBER\_OCT  
 elif char in ("8", "9"):  
 yield States.NUMBER\_DEC  
 elif char in ("e", "E"):  
 yield States.LETTER\_E  
 elif char == ".":  
 yield States.FRACTIONAL  
 elif char in ("b", "B"):  
 yield States.LETTER\_B  
 elif char in ("o", "O"):  
 yield States.LETTER\_O  
 elif char in ("d", "D"):  
 yield States.LETTER\_D  
 elif char in SEPARATORS:  
 yield Lex.NUMBER\_DEC, States.START  
 elif char in ("a", "b", "c", "d", "e", "f", "A", "B", "C", "D", "E", "F"):  
 yield States.NUMBER\_HEX  
 elif char in ("h", "H"):  
 yield States.LETTER\_H  
 else:  
 yield ["Число не может быть построено, некорректный символ в числе", States.ER]  
  
  
def state\_letter\_b\_handler():  
 while True:  
 char: str = yield  
  
 if char in SEPARATORS:  
 yield Lex.NUMBER\_BIN, States.START  
 elif char.isdigit() or char in ("a", "b", "c", "d", "e", "f", "A", "B", "C", "D", "E", "F"):  
 yield States.NUMBER\_HEX  
 elif char in ("h", "H"):  
 yield States.LETTER\_H  
 else:  
 yield ["После двоичного числа необходим разделитель", States.ER]  
  
  
def state\_letter\_d\_hander():  
 while True:  
 char: str = yield  
  
 if char in SEPARATORS:  
 yield Lex.NUMBER\_DEC, States.START  
 elif char.isdigit() or char in ("a", "b", "c", "d", "e", "f", "A", "B", "C", "D", "E", "F"):  
 yield States.NUMBER\_HEX  
 elif char in ("h", "H"):  
 yield States.LETTER\_H  
 else:  
 yield ["После десятичного числа необходим разделитель", States.ER]  
  
  
def state\_letter\_e\_hander():  
 # 10E0H  
 # 10E10  
 # 10E+10  
 # 10EH  
 """  
 state = 0 - не можем определить, что будет в итоге  
 state = 1 - формируется либо 16-ричное число, либо порядок числа  
 state = 2 - порядок числа  
 """  
  
 state = 0  
  
 while True:  
 char: str = yield  
  
 if state == 0:  
 if char in ("+", "-"):  
 yield States.LETTER\_E  
 state = 2  
 elif char in ("a", "b", "c", "d", "e", "f", "A", "B", "C", "D", "E", "F"):  
 yield States.NUMBER\_HEX  
 elif char in ("h", "H"):  
 yield States.LETTER\_H  
 elif char.isdigit():  
 yield States.LETTER\_E  
 state = 1  
 else:  
 yield ["Шестнадцатиричное число или порядок дробного записаны некорректно", States.ER]  
 elif state == 1:  
 if char.isdigit():  
 yield States.LETTER\_E  
 elif char in ("h", "H"):  
 yield States.LETTER\_H  
 elif char in ("a", "b", "c", "d", "e", "f", "A", "B", "C", "D", "E", "F"):  
 yield States.NUMBER\_HEX  
 elif char in SEPARATORS:  
 yield Lex.NUMBER\_FRACTIONAL, States.START  
 else:  
 yield ["Шестнадцатиричное число или порядок дробного записаны некорректно", States.ER]  
 elif state == 2:  
 if char.isdigit():  
 yield States.LETTER\_E  
 elif char in SEPARATORS:  
 yield Lex.NUMBER\_FRACTIONAL, States.START  
 else:  
 yield ["Порядок дробного числа записан некорректно", States.ER]  
  
  
def state\_letter\_h\_handler():  
 while True:  
 char: str = yield  
 if char in SEPARATORS:  
 yield Lex.NUMBER\_HEX, States.START  
 else:  
 yield ["После шестнадцатиричного числа необходим разделитель", States.ER]  
  
  
def state\_letter\_o\_handler():  
 while True:  
 char: str = yield  
 if char in SEPARATORS:  
 yield Lex.NUMBER\_OCT, States.START  
 else:  
 yield ["После восьмиричного числа необходим разделитель", States.ER]  
  
  
def state\_number\_oct\_handler():  
 while True:  
 char: str = yield  
  
 if char in ("0", "1", "2", "3", "4", "5", "6", "7"):  
 yield States.NUMBER\_OCT  
 elif char in ("8", "9"):  
 yield States.NUMBER\_DEC  
 elif char in ("e", "E"):  
 yield States.LETTER\_E  
 elif char == ".":  
 yield States.FRACTIONAL  
 elif char in ("o", "O"):  
 yield States.LETTER\_O  
 elif char in ("d", "D"):  
 yield States.LETTER\_D  
 elif char in SEPARATORS:  
 yield Lex.NUMBER\_DEC, States.START  
 elif char in ("h", "H"):  
 yield States.LETTER\_H  
 elif char in ("a", "b", "c", "d", "e", "f", "A", "B", "C", "D", "E", "F"):  
 yield States.NUMBER\_HEX  
 else:  
 yield ["Число не может быть построено, некорректный символ в числе", States.ER]  
  
  
def state\_number\_dec\_handler():  
 while True:  
 char: str = yield  
  
 if char.isdigit():  
 yield States.NUMBER\_DEC  
 elif char in ("e", "E"):  
 yield States.LETTER\_E  
 elif char == ".":  
 yield States.FRACTIONAL  
 elif char in ("d", "D"):  
 yield States.LETTER\_D  
 elif char in ("h", "H"):  
 yield States.LETTER\_H  
 elif char in ("a", "b", "c", "d", "e", "f", "A", "B", "C", "D", "E", "F"):  
 yield States.NUMBER\_HEX  
 elif char in SEPARATORS:  
 yield Lex.NUMBER\_DEC, States.START  
 else:  
 yield ["Число не может быть построено, некорректный символ в числе", States.ER]  
  
  
def state\_number\_hex\_handler():  
 while True:  
 char: str = yield  
  
 if char.isdigit() or char in ("a", "b", "c", "d", "e", "f", "A", "B", "C", "D", "E", "F"):  
 yield States.NUMBER\_HEX  
 elif char in ("h", "H"):  
 yield States.LETTER\_H  
 else:  
 yield ["Число не может быть построено, один из символов выходит за рамки числа", States.ER]  
  
  
def state\_fractional\_handler():  
 state = 0  
 """  
 state = 0 - пока не пуступила ни 1-на цифра  
 state = 1 - поступила как минимум 1-на цифра  
 """  
 while True:  
 char: str = yield  
  
 if state == 0:  
 if char.isdigit():  
 yield States.FRACTIONAL  
 state = 1  
 else:  
 yield ["Дробное число: после символа '.' необходима цифра", States.ER]  
 elif state == 1:  
 if char.isdigit():  
 yield States.FRACTIONAL  
 elif char in ("e", "E"):  
 yield States.NUMBER\_ORDER  
 elif char in SEPARATORS:  
 yield Lex.NUMBER\_FRACTIONAL, States.START  
 else:  
 yield ["Дробное число: после числа необходим порядок или разделитель", States.ER]  
  
  
def state\_number\_order\_handler():  
 state = 0  
 while True:  
 char: str = yield  
 if state == 0:  
 if char in ("+", "-"):  
 yield States.NUMBER\_ORDER  
 state = 1  
 elif char.isdigit():  
 yield States.NUMBER\_ORDER  
 state = 2  
 else:  
 yield ["Порядок записан некорректно", States.ER]  
 elif state == 1:  
 if char.isdigit():  
 yield States.NUMBER\_ORDER  
 state = 2  
 else:  
 yield ["Порядок записан некорректно", States.ER]  
 elif state == 2:  
 if char.isdigit():  
 yield States.NUMBER\_ORDER  
 elif char in SEPARATORS:  
 yield Lex.NUMBER\_FRACTIONAL, States.START  
 else:  
 yield ["Порядок записан некорректно", States.ER]  
  
  
def state\_er\_handler():  
 while True:  
 char: str = yield  
 if char in SEPARATORS:  
 yield Lex.UNRESOLVED, States.START  
 else:  
 yield States.ER  
  
  
HANDLERS = {  
 States.START: state\_start\_handler,  
 States.IDENTIFIER: state\_identificator\_handler,  
 States.NUMBER\_BIN: state\_number\_bin\_handler,  
 States.NUMBER\_OCT: state\_number\_oct\_handler,  
 States.NUMBER\_DEC: state\_number\_dec\_handler,  
 States.NUMBER\_HEX: state\_number\_hex\_handler,  
 States.FRACTIONAL: state\_fractional\_handler,  
 States.NUMBER\_ORDER: state\_number\_order\_handler,  
 States.LETTER\_B: state\_letter\_b\_handler,  
 States.LETTER\_D: state\_letter\_d\_hander,  
 States.LETTER\_E: state\_letter\_e\_hander,  
 States.LETTER\_H: state\_letter\_h\_handler,  
 States.LETTER\_O: state\_letter\_o\_handler,  
 States.SEPARATOR\_NOT: state\_separator\_not\_handler,  
 States.SEPARATOR\_EQUALS: state\_separator\_equals\_handler,  
 States.SEPARATOR\_LT: state\_separator\_lt\_handler,  
 States.SEPARATOR\_GT: state\_separator\_gt\_handler,  
 States.SEPARATOR\_OR: state\_separator\_or\_handler,  
 States.SEPARATOR\_AND: state\_separator\_and\_handler,  
 States.SEPARATOR\_ASSIGNMENT: state\_separator\_assignment\_handler,  
 States.SEPARATOR\_COMMENT: state\_separator\_comment\_handler,  
 States.ER: state\_er\_handler,  
}  
  
  
class HandlerFactory:  
 """  
 Класс фабрика,  
 позволяет получать нужный обработчик без необходимости его обновления  
 """  
  
 # кэшированные генераторы  
 \_cache: dict  
  
 def \_\_init\_\_(self):  
 self.\_cache = {}  
  
 def get\_handler(self, state: States):  
 """  
 Метод возвращает генератор из кэша или возвращает созданный, обернув его в оболочку renewable  
 в зависимости от состояния  
 """  
 from\_cache = self.\_cache.get(state, None)  
 if (from\_cache is not None):  
 return from\_cache  
 handler = self.init\_handler(state)  
 self.\_cache[state] = handler  
 return handler  
  
 def init\_handler(self, state):  
 """  
 Метод инициализирует генератор-обработчик состояния, оборачивает его в renewable,  
 инициализирует renewable и возвращает его  
 """  
 global HANDLERS  
 handler = HANDLERS.get(state)()  
 handler.send(None)  
 handler = self.renewable(state, handler)  
 handler.send(None)  
 return handler  
  
 def renewable(self, state: States, activated\_generator: state\_handler):  
 """  
 Генератор-делегатор,  
 следит за тем, когда подгенератор activated\_generator возвращает состояния, соответствующие концу работы генератора  
 и кладёт в кэш новый инициализированный объект генератор, чтобы сбросить внутренние состояния для следующих лексем  
 """  
  
 def wrapper(\*args, \*\*kwargs):  
 new\_state = None  
 while True:  
 data = yield new\_state  
 new\_state = activated\_generator.send(data)  
 if new\_state != state:  
 self.\_cache[state] = self.init\_handler(state)  
 next(activated\_generator)  
  
 return wrapper()

Листинг 11.17 – src/lang/\_\_init\_\_.py

from .lang\_base\_types import \*

Листинг 11.18 – src/lang/lang\_base\_types.py

from abc import ABC, abstractmethod  
from enum import Enum  
from typing import Union  
  
from src.constants import Lex  
  
  
class BaseType(ABC):  
 value: Union[int, float, bool]  
  
 @staticmethod  
 @abstractmethod  
 def from\_string(literal: str):  
 pass  
  
 def neq(self, other):  
 return Boolean(self.value.\_\_ne\_\_(other.value))  
  
 def eq(self, other):  
 return Boolean(self.value.\_\_eq\_\_(other.value))  
  
 def \_\_str\_\_(self):  
 return str(self.value)  
  
  
class Number(BaseType):  
  
 @abstractmethod  
 def add(self, other):  
 pass  
  
 @abstractmethod  
 def sub(self, other):  
 pass  
  
 @abstractmethod  
 def mul(self, other):  
 pass  
  
 @abstractmethod  
 def div(self, other):  
 pass  
  
 def lt(self, other):  
 tmp = self.value.\_\_lt\_\_(other.value)  
 return Boolean(self.value.\_\_lt\_\_(other.value))  
  
 def lte(self, other):  
 return Boolean(self.value.\_\_le\_\_(other.value))  
  
 def gt(self, other):  
 return Boolean(self.value.\_\_gt\_\_(other.value))  
  
 def gte(self, other):  
 return Boolean(self.value.\_\_ge\_\_(other.value))  
  
  
class Integer(Number):  
 @staticmethod  
 def from\_string(literal: str):  
 if literal.endswith(("b", "B")):  
 return Integer(int(literal[:-1], 2))  
 elif literal.endswith(("o", "O")):  
 return Integer(int(literal[:-1], 8))  
 elif literal.endswith(("h", "H")):  
 return Integer(int(literal[:-1], 16))  
 elif literal.endswith(("d", "D")):  
 return Integer(int(literal[:-1]))  
 return Integer(int(literal))  
  
 def \_\_init\_\_(self, value: int):  
 if not isinstance(value, int):  
 raise ValueError("Значение параметра value должно быть типа int")  
 self.value = value  
  
 def add(self, other):  
 return Integer(self.value + other.value)  
  
 def sub(self, other):  
 return Integer(self.value - other.value)  
  
 def mul(self, other):  
 return Integer(self.value \* other.value)  
  
 def div(self, other):  
 return Integer(self.value // other.value)  
  
 def mod(self, other):  
 return Integer(self.value % other.value)  
  
  
class Float(Number):  
 @staticmethod  
 def from\_string(literal: str):  
 return Float(float(literal))  
  
 def \_\_init\_\_(self, value: float):  
 if not isinstance(value, float):  
 raise ValueError("Значение параметра value должно быть типа float")  
 self.value = value  
  
 def add(self, other):  
 return Float(self.value + other.value)  
  
 def sub(self, other):  
 return Float(self.value - other.value)  
  
 def mul(self, other):  
 return Float(self.value \* other.value)  
  
 def div(self, other):  
 return Float(self.value / other.value)  
  
  
class Boolean(BaseType):  
 @staticmethod  
 def from\_string(literal: str):  
 if literal == "true":  
 return Boolean(True)  
 return Boolean(False)  
  
 def \_\_init\_\_(self, value: bool):  
 if not isinstance(value, bool):  
 raise ValueError("Значение параметра value должно быть типа bool")  
 self.value = value  
  
 def or\_(self, other):  
 return Boolean(bool(self.value) or bool(other.value))  
  
 def and\_(self, other):  
 return Boolean(bool(self.value) and bool(other.value))  
  
 def not\_(self):  
 return Boolean(not bool(self.value))  
  
  
TYPES = {  
 Lex.KEYWORD\_INT: Integer,  
 Lex.KEYWORD\_FLOAT: Float,  
 Lex.KEYWORD\_BOOL: Boolean  
}  
  
  
class Types(Enum):  
 int = 0  
 float = 1  
 bool = 2  
  
  
LEX\_TO\_TYPE\_TABLE = {  
 Lex.KEYWORD\_INT: Types.int,  
 Lex.KEYWORD\_FLOAT: Types.float,  
 Lex.KEYWORD\_BOOL: Types.bool  
}  
  
TYPE\_TO\_TYPE\_TABLE = {  
 Integer: Types.int,  
 Float: Types.float,  
 Boolean: Types.bool  
}  
  
  
def type\_to\_type(t\_type: Union[Integer, Float, Boolean]):  
 r = TYPE\_TO\_TYPE\_TABLE.get(t\_type.\_\_class\_\_)  
 if r is not None:  
 return r  
 raise ValueError("Тип не определён")  
  
  
def get\_type\_number\_from\_lex(lex: Lex) -> Types:  
 r = LEX\_TO\_TYPE\_TABLE.get(lex)  
 if r is not None:  
 return r  
 raise ValueError("Тип не определён")  
  
  
class BinOperations(Enum):  
 sum = 0 # сумма  
 diff = 1 # разность  
 mul = 2 # произведение  
 div = 3 # частное  
 alt = 4 # операция или  
 con = 5 # операция и  
 eq = 6 # ==  
 neq = 7 # !=  
 gt = 8 # >  
 gte = 9 # >=  
 lt = 10 # <  
 lte = 11 # <=  
  
 # Оптимизация дерева  
 mod = 12 # %  
  
  
RELATION\_OPERATORS = [  
 BinOperations.lt,  
 BinOperations.lte,  
 BinOperations.gt,  
 BinOperations.gte,  
 BinOperations.eq,  
 BinOperations.neq  
]  
  
LEX\_TO\_BIN\_OP\_TABLE = {  
 Lex.SEPARATOR\_PLUS: BinOperations.sum,  
 Lex.SEPARATOR\_MINUS: BinOperations.diff,  
 Lex.SEPARATOR\_MULTIPLICATION: BinOperations.mul,  
 Lex.SEPARATOR\_DIVISION: BinOperations.div,  
 Lex.SEPARATOR\_OR: BinOperations.alt,  
 Lex.SEPARATOR\_AND: BinOperations.con,  
 Lex.SEPARATOR\_EQUALS: BinOperations.eq,  
 Lex.SEPARATOR\_NOT\_EQUALS: BinOperations.neq,  
 Lex.SEPARATOR\_GT: BinOperations.gt,  
 Lex.SEPARATOR\_GTE: BinOperations.gte,  
 Lex.SEPARATOR\_LT: BinOperations.lt,  
 Lex.SEPARATOR\_LTE: BinOperations.lte  
}  
  
  
def get\_bin\_operation\_from\_lex(lex: Lex) -> BinOperations:  
 r = LEX\_TO\_BIN\_OP\_TABLE.get(lex)  
 if r is not None:  
 return r  
 raise ValueError("Операция не определён")

Листинг 11.19 – src/cpp\_translator/\_\_init\_\_.py

from .translator import translate

Листинг 11.20 – src/cpp\_translator/translator.py

"""  
Модуль синтезации программы на c++ из абстрактного дерева  
"""  
  
from src.tree import \*  
  
type\_enum\_to\_type\_str\_table = {  
 Types.int: "long long",  
 Types.float: "double",  
 Types.bool: "bool"  
}  
  
python\_bool\_to\_cpp\_bool = {  
 True: "true",  
 False: "false"  
}  
  
def translate(ast: ASTNode):  
 r = "#include <iostream>\n\n"  
 r += "int main()\n{\n"  
 while ast is not None:  
 r += translate\_operator(ast)  
 ast = ast.next\_node  
 r += "\treturn 0;\n}\n"  
 return r  
  
  
def translate\_expression(ast: ASTTyped) -> str:  
 if ast.a\_type == ASTType.CONST:  
 ast: ASTConst  
 if ast.t\_type == Types.bool:  
 return python\_bool\_to\_cpp\_bool[ast.value.value]  
 return str(ast.value)  
 elif ast.a\_type == ASTType.VAR:  
 ast: ASTVar  
 return str(ast.name)  
  
 elif ast.a\_type == ASTType.U\_OP:  
 ast: ASTUOperation  
 return "(!" + translate\_expression(ast.operand) + ")"  
 elif ast.a\_type == ASTType.BIN\_OP:  
 ast: ASTBinOperation  
 if ast.operation == BinOperations.sum:  
 return "(" + translate\_expression(ast.left) + " + " + translate\_expression(ast.right) + ")"  
 elif ast.operation == BinOperations.diff:  
 return "(" + translate\_expression(ast.left) + " - " + translate\_expression(ast.right) + ")"  
 elif ast.operation == BinOperations.mul:  
 return "(" + translate\_expression(ast.left) + " \* " + translate\_expression(ast.right) + ")"  
 elif ast.operation == BinOperations.div:  
 return "(" + translate\_expression(ast.left) + " / " + translate\_expression(ast.right) + ")"  
 elif ast.operation == BinOperations.alt:  
 return "(" + translate\_expression(ast.left) + " || " + translate\_expression(ast.right) + ")"  
 elif ast.operation == BinOperations.con:  
 return "(" + translate\_expression(ast.left) + " && " + translate\_expression(ast.right) + ")"  
 elif ast.operation == BinOperations.lt:  
 return "(" + translate\_expression(ast.left) + " < " + translate\_expression(ast.right) + ")"  
 elif ast.operation == BinOperations.lte:  
 return "(" + translate\_expression(ast.left) + " <= " + translate\_expression(ast.right) + ")"  
 elif ast.operation == BinOperations.gt:  
 return "(" + translate\_expression(ast.left) + " > " + translate\_expression(ast.right) + ")"  
 elif ast.operation == BinOperations.gte:  
 return "(" + translate\_expression(ast.left) + " >= " + translate\_expression(ast.right) + ")"  
 elif ast.operation == BinOperations.eq:  
 return "(" + translate\_expression(ast.left) + " == " + translate\_expression(ast.right) + ")"  
 elif ast.operation == BinOperations.neq:  
 return "(" + translate\_expression(ast.left) + " != " + translate\_expression(ast.right) + ")"  
 elif ast.operation == BinOperations.mod:  
 return "(" + translate\_expression(ast.left) + " % " + translate\_expression(ast.right) + ")"  
 else:  
 raise ValueError("Неопределённая операция")  
  
  
def translate\_operator(ast: ASTNode, tabs: int = 1):  
 if ast.a\_type == ASTType.DECL:  
 ast: ASTDeclaration  
 r = "\t" \* tabs + type\_enum\_to\_type\_str\_table.get(ast.t\_type)  
 node = ast.variables  
 r += " " + node.name  
 node = node.next\_node  
 while node is not None:  
 r += ", " + node.name  
 node = node.next\_node  
 r += ";\n"  
 return r  
  
 elif ast.a\_type == ASTType.ASSIGNMENT:  
 ast: ASTAssignment  
 r = "\t" \* tabs + ast.var.name + " = " + translate\_expression(ast.value)  
 r += ";\n"  
 return r  
  
 elif ast.a\_type == ASTType.IF:  
 ast: ASTIf  
 condition = translate\_expression(ast.condition)  
 r = "\t" \* tabs + "if (" + condition + ")" + "\n" + "\t" \* tabs + "{\n"  
 node = ast.branch  
 while node is not None:  
 r += translate\_operator(node, tabs + 1)  
 node = node.next\_node  
 r += "\t" \* tabs + "}\n"  
 if ast.else\_branch is not None:  
 r += "\t" \* tabs + "else\n" + "\t" \* tabs + "{\n"  
 node = ast.else\_branch  
 while node is not None:  
 r += translate\_operator(node, tabs + 1)  
 node = node.next\_node  
 r += "\t" \* tabs + "}\n"  
 return r  
  
 elif ast.a\_type == ASTType.Loop:  
 ast: ASTLoop  
 condition = translate\_expression(ast.condition)  
 r = "\t" \* tabs + "while (" + condition + ")" + "\n" + "\t" \* tabs + "{\n"  
 node = ast.body  
 while node is not None:  
 r += translate\_operator(node, tabs + 1)  
 node = node.next\_node  
 r += "\t" \* tabs + "}\n"  
 return r  
  
 elif ast.a\_type == ASTType.ForLoop:  
 ast: ASTForLoop  
 assignment = type\_enum\_to\_type\_str\_table.get(  
 Types.int) + " " + ast.assignment.var.name + " = " + translate\_expression(ast.assignment.value)  
 condition = translate\_expression(ast.condition)  
 step = translate\_expression(ast.step)  
 r = ("\t" \* tabs + "for (" + assignment + "; " + condition + "; " + ast.assignment.var.name +  
 " += " + step + ")" + "\n" + "\t" \* tabs + "{\n")  
 node = ast.body  
 while node is not None:  
 r += translate\_operator(node, tabs + 1)  
 node = node.next\_node  
  
 r += "\t" \* tabs + "}\n"  
 return r  
  
 elif ast.a\_type == ASTType.IN:  
 ast: ASTIn  
 node = ast.variables  
 r = ""  
 while node is not None:  
 r += "\t" \* tabs + "std::cin >> " + node.name + ";\n"  
 node = node.next\_node  
 return r  
  
 elif ast.a\_type == ASTType.OUT:  
 ast: ASTOut  
 node = ast.expressions  
 r = ""  
 while node is not None:  
 r += "\t" \* tabs + "std::cout << " + translate\_expression(node) + " << std::endl;\n"  
 node = node.next\_node  
 return r  
 else:  
 raise ValueError("Оператор не определён")

# Приложение Б — Контрольный пример

Контрольный пример

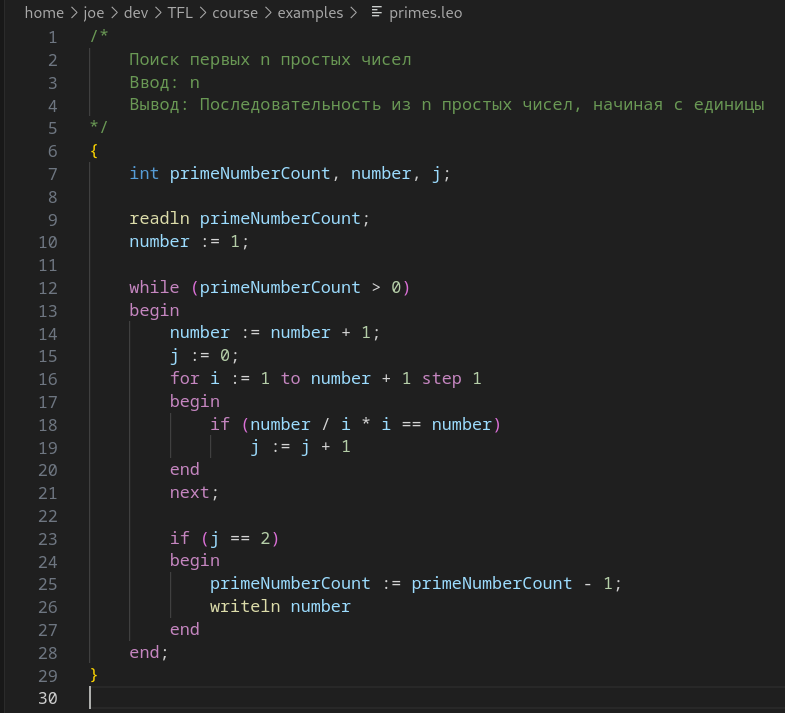


Рисунок 12.1 – Программа для поиска первых primeNumberCount простых чисел

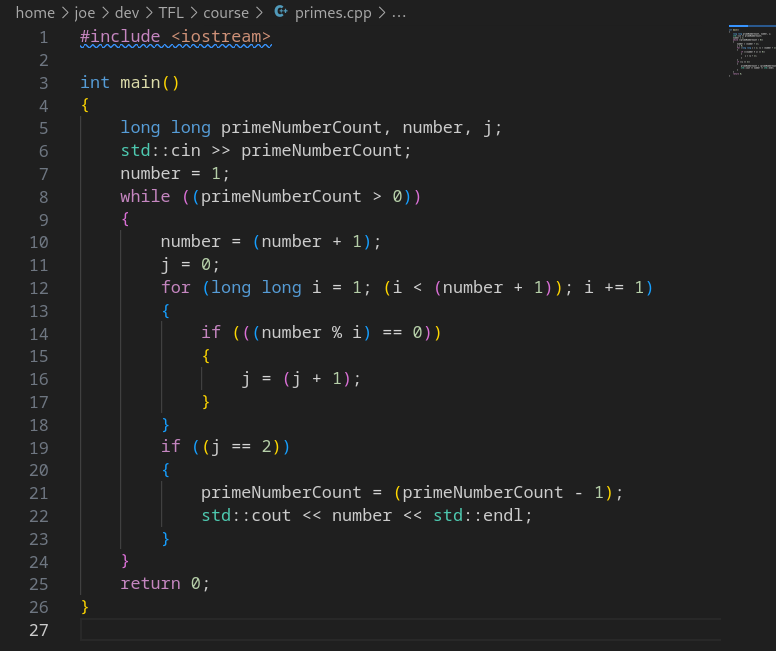


Рисунок 12.2 – Результат трансляции на c++

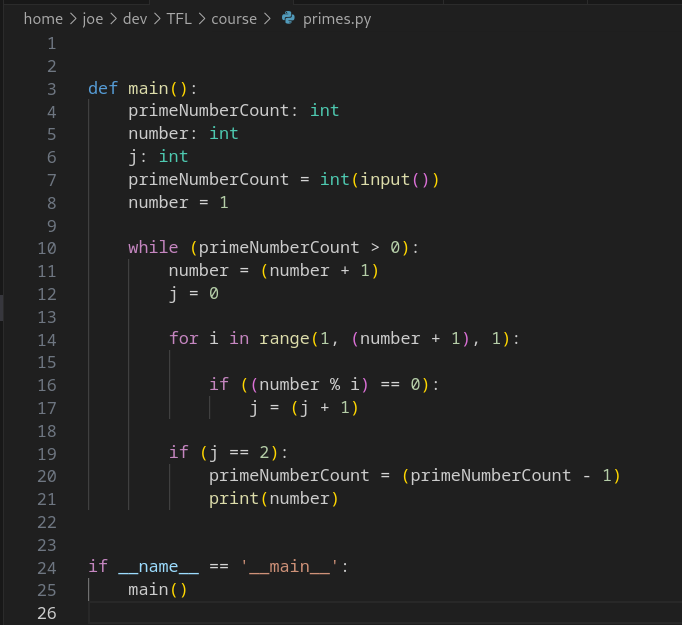


Рисунок 12.3 – Результат трансляции на Python

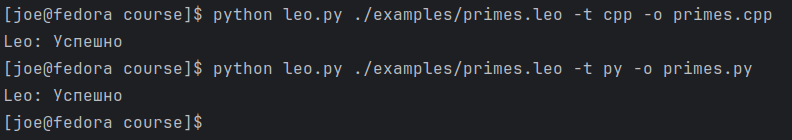


Рисунок 12.4 – Вывод в консоль

# Приложение В — Сообщения об ошибках

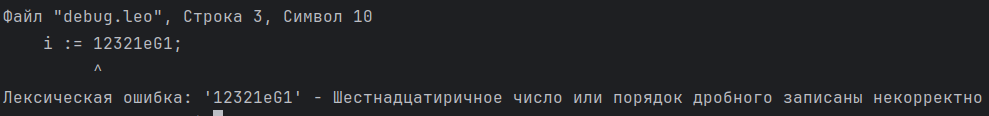


Рисунок 13.1 – Лексическая ошибка

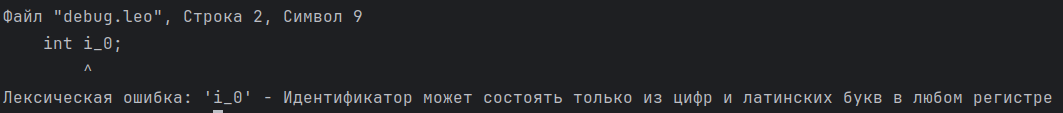


Рисунок 13.2 – Лексическая ошибка

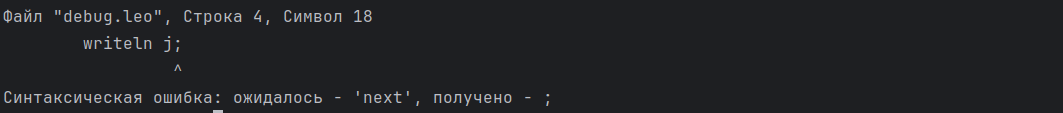


Рисунок 13.3 – Синтаксическая ошибка

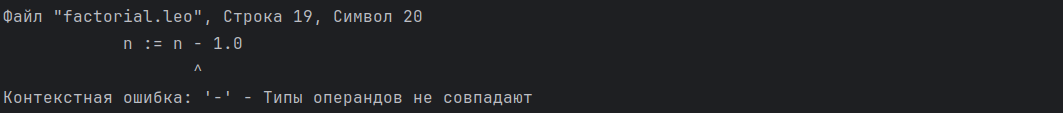


Рисунок 13.4 – Семантическая ошибка