МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное автономное образовательное учреждение  
высшего образования

**«Дальневосточный федеральный университет»**

**ИНСТИТУТ МАТЕМАТИКИ И КОМПЬЮТЕРНЫХ ТЕХНОЛОГИЙ**

**Департамент программной инженерии и искусственного интеллекта**

РАЗРАБОТКА ТРАНСЛЯТОРА, ПЕРЕВОДЯЩЕГО ПОДМНОЖЕСТВО ЯЗЫКА PYTHON В ЭКВИВАЛЕНТНОЕ ПОДМНОЖЕСТВО ЯЗЫКА LUA

**ОТЧЕТ ПО ЛАБОРАТОРНОЙ РАБОТЕ**

по дисциплине «Теория языков программирования и компиляторы»

по образовательной программе подготовки бакалавров по направлению 09.03.04 Программная инженерия

Выполнили студенты гр. Б9118-09.03.04прогин

\_\_\_\_\_\_\_\_\_\_\_ Аксененко О.Ю.

\_\_\_\_\_\_\_\_\_\_\_ Говоров В.Р.

\_\_\_\_\_\_\_\_\_\_\_ Жлуткин Р.В.

\_\_\_\_\_\_\_\_\_\_\_ Рева С.Б.

\_\_\_\_\_\_\_\_\_\_\_ Шешин М.С.

Руководитель: ассистент Шутов К. С.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_ «\_\_\_» \_\_\_\_\_\_\_\_\_\_\_\_ 2021 г.

(подпись)

Защищён с оценкой \_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_

(подпись) И.О. Фамилия

«\_\_\_» \_\_\_\_\_\_\_\_\_\_ 2021 г.

г. Владивосток

2021 г.

Оглавление

[Введение 3](#_Toc87452605)

[1 Теоретическая часть 4](#_Toc87452606)

[2 Неформальная постановка задачи 7](#_Toc87452607)

[3 Требования к системе 8](#_Toc87452608)

[4 Синтаксис входного языка 9](#_Toc87452609)

[5 Контекстные условия 13](#_Toc87452610)

[6 Проект системы 14](#_Toc87452611)

[7 Проект лексического анализатора 15](#_Toc87452612)

[7.1 Таблица ключевых слов 16](#_Toc87452613)

[7.2 Таблица зарезервированных имён 16](#_Toc87452614)

[7.3 Типы лексем 17](#_Toc87452615)

[7.4 Представление лексического анализатора 18](#_Toc87452616)

[7.5 Ошибки лексического анализатора 19](#_Toc87452617)

[7.6 Работа лексического анализатора 19](#_Toc87452618)

[7.7 Модель данных 20](#_Toc87452619)

[8 Проект синтаксического анализатора 21](#_Toc87452620)

[8.1 Описание алгоритма 22](#_Toc87452621)

[8.2 Модель данных 23](#_Toc87452622)

[8.3 Полный текст грамматики в компьютерно-читаемом виде 25](#_Toc87452623)

[8.4 Проект алгоритма 28](#_Toc87452624)

[8.5 Работа синтаксического анализатора 28](#_Toc87452625)

[8.6 Ошибки синтаксического анализатора 30](#_Toc87452626)

[9 Проект семантического анализатора 30](#_Toc87452627)

[9.1 Работа семантического анализатора 30](#_Toc87452628)

[9.2 Ошибки семантического анализатора 31](#_Toc87452629)

[10 Проект генератора кода 32](#_Toc87452630)

[10.1 Модель данных 32](#_Toc87452631)

[10.2 Таблица соответствия 32](#_Toc87452632)

[11 Тестирование 34](#_Toc87452633)

[11.1 Тестирование лексического анализатора 34](#_Toc87452634)

[11.2 Тестирование синтаксического анализатора 35](#_Toc87452635)

[11.3 Тестирование семантического анализатора 38](#_Toc87452636)

[11.4 Тестирование генератора кода 41](#_Toc87452637)

[Заключение 44](#_Toc87452638)

[Список литературы 45](#_Toc87452639)

# Введение

Компилятор – это программа, которая считывает текст программы, написанной на одном языке – исходном, и переводит его в эквивалентный текст на другом языке – целевом.

Отображение исходной программы в семантически эквивалентную ей целевую программу разделяется на две части: анализ и синтез.

Анализ разбивает исходную программу на составные части и накладывает на них грамматическую структуру. Затем он использует эту структуру для создания промежуточного представления исходной программы. Если анализ обнаруживает, что исходная программа неверно составлена синтаксически либо дефектна семантически, он должен выдать информативные сообщения об этом. Анализ также собирает информацию об исходной программе и сохраняет её в структуре данных, именуемой таблицей символов, которая передаётся вместе с промежуточным представлением синтезу.

Синтез строит требуемую целевую программу на основе промежуточного представления и информации из таблицы символов [1].

Цель работы: разработать транслятор, который преобразует программу, содержащую подмножество языка Python, в программу, содержащую подмножество языка Lua, генерируя эквивалентный исходный код.

Задачи:

* разработать грамматику подмножества языка Python;
* описать контекстные условия входного языка;
* описать соответствие конструкций входного и выходного языков;
* разработать проект лексического анализатора;
* разработать проект синтаксического анализатора;
* разработать проект семантического анализатора;
* разработать проект генератора кода;
* реализовать разработанные проекты в виде программы-транслятора.

# 1 Теоретическая часть

**Алфавитом** (или **словарём**) называется любое непустое конечное множество *Σ*. Элементы этого множества называются **символами** (**буквами** или **знаками**) [2].

Произвольная конечная последовательность элементов *Σ* называется **цепочкой** в алфавите *Σ* [2].

**Языком** в алфавите *Σ* называется произвольное множество цепочек в *Σ* [2].

Существует несколько способов задания языков:

1. явное задание языка (перечисляется всё множество цепочек языка);

2. задание языка регулярным выражением;

3. задание языка при помощи порождающей грамматики;

4. задание языка при помощи распознавателя.

**Регулярные выражения** в алфавите *Σ* и **регулярные множества**, которые они обозначают, определяются рекурсивно следующим образом:

1. ∅ – регулярное выражение, обозначающее регулярное множество ∅;

2. *λ* – регулярное выражение, обозначающее множество {*λ*};

3. если *а* ∈ *Σ*, то а – регулярное выражение, обозначающее регулярное множество {*а*};

4. если *p* и *q* – регулярные выражения, обозначающие регулярные множества *P* и *Q* соответственно, то

а. (*p* + *q*) – регулярное выражение, обозначающее регулярное множество *P* ∪ *Q*;

б) (*pq*) – регулярное выражение, обозначающее регулярное множество *PQ*;

в) (*p*)\* – регулярное выражение, обозначающее *P*\*;

5. ничто другое не является регулярным выражением [2].

**Порождающей грамматикой** называется четвёрка *G = (N, Σ, P, S)*, где:

* *N* – нетерминальный (вспомогательный) словарь – конечное (непустое) множество нетерминальных символов, или нетерминалов (иногда называемых вспомогательными символами, синтаксическими переменными или понятиями);
* *Σ* – терминальный словарь – не пересекающееся с *N* конечное (непустое) множество терминальных символов, или терминалов;
* *P* – схема грамматики – конечное подмножество множества *(N* ∪ *Σ)\* N (N* ∪ *Σ) × (N* ∪ *Σ)\**; элемент (*α*, *β*) ∈ *P*, где *α* ∈ (*N* ∪ *Σ*)\* *N* (*N* ∪ *Σ*)\*, а *β* ∈ (*N* ∪ *Σ*)\* называется правилом (или продукцией) и записывается в виде *α* → *β*, *α* называется левой частью правила, а *β* – правой частью правила);
* *S* – выделенный символ из *N*, называемый начальным (исходным) символом, или аксиомой грамматики [2].

**Распознаватель** – схематизированный алгоритм, определяющий некоторое множество. Распознаватель можно изобразить так, как показано на рисунке 1 [2].

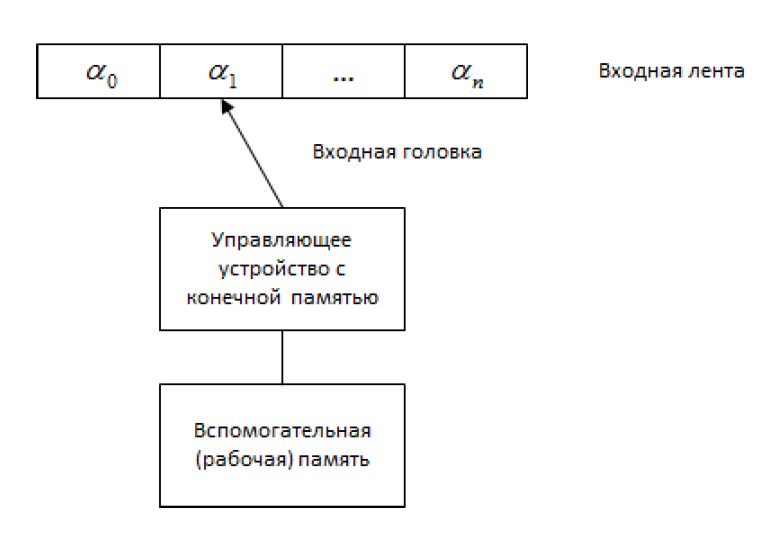


Рисунок 1 – Распознаватель

Распознаватель состоит из трёх частей: входная лента, управляющее устройство с конечной памятью, вспомогательная (или рабочая) память.

**Язык, определяемый распознавателем** – множество входных цепочек, которые он допускает [2].

Таким образом, любой распознаватель характеризуется:

1) множеством состояний управляющего устройства;

2) алфавитом входной ленты;

3) алфавитом рабочей памяти (если рабочая память является компонентом распознавателя);

4) функцией переходов, описывающей правила изменения состояний рабочей памяти и управляющего устройства в зависимости от текущего состояния входной ленты, управляющего устройства и рабочей памяти;

5) начальным состоянием рабочей памяти;

6) начальным состоянием управляющего устройства;

7) множеством заключительных состояний управляющего устройства.

Существует два типа распознавателей: конечные автоматы (модель лексического анализатора) и автоматы с магазинной памятью (модель синтаксического анализатора).

Процесс **трансляции** состоит из следующих фаз:

1. лексический анализ;
2. синтаксический анализ;
3. семантический анализ;
4. генерация кода.

Основная задача **лексического анализа** – разбиение входного текста, состоящего из последовательности одиночных символов, на последовательность слов, или лексем, т.е. выделение этих слов из непрерывной последовательности символов. Все символы входной последовательности с этой точки зрения разделяются на символы, принадлежащие каким-либо лексемам, и символы, разделяющие лексемы.

К числу дополнительных задач лексического анализатора можно отнести удаление комментариев из текста программы и не играющих никакой роли в программе пробелов, а также поиск некоторых ошибок.

Задачей **синтаксического анализа** является проверка правильности структуры программы, которая описывается контекстно-свободной грамматикой. Класс контекстно-свободных грамматик шире класса автоматных грамматик, поэтому и синтаксический анализатор устроен сложнее лексического.

**Семантический анализ** проверяет контекстные условия языка. Контекстные условия языков программирования формулируются с помощью естественного языка. Ими могут быть условия о правилах описания идентификаторов в программах; условия о правилах использования идентификаторов в своей области действия; условия, определяющие правила соответствия видов величин, входящих в синтаксические конструкции программ; условия, задающие различные количественные ограничения и другие [3].

**Генератор кода** осуществляет преобразование синтаксически корректной программы в объектный код.

# 2 Неформальная постановка задачи

Разработать транслятор из подмножества языка Python в эквивалентное подмножество языка Lua.

Подмножество языка Python включает:

1. Типы данных: поскольку Python характеризуется сильной динамической типизацией, типы данных для этого языка большого значения не имеют: информация о типе данных содержится только в самой переменной и используется лишь для вызова методов, связанных с ней. Поле с типом данных, на который ссылается переменная, легко меняется, что, например, позволяет произвести присваивание переменной сначала целочисленного значения, а затем строкового.

2. Операторы: +, -, \*, /, %, \*\*, ==, !=, >, <, >=, <=, and, or, not, in, =.

3. Операторы циклов: for, while.

4. Операторы ветвления: if, elif, else.

5. Стандартные функции: print, len, dict, range.

6. Логические константы: True, False.

# 3 Требования к системе

1. **Функциональные требования:**
   * программа-транслятор должна создавать файл с текстом программы на языке, принадлежащему подмножеству языка Lua, на основе файла с текстом программы на эквивалентном подмножестве языка Python, при условии, что текст программы допустим описанной грамматикой;
   * программа должна работать в операционных системах Windows, Linux, MacOS;
   * результат выполнения исходной программы должен совпадать с результатом исполнения объектной программы;
   * программа должна выполнять лексический, синтаксический и семантический анализ текста исходной программы из входного файла, при этом если один из анализаторов обнаруживает ошибку, то пользователю выводится сообщение об ошибке и работа программы-транслятора завершается.
2. **Требования к входным данным:**
   * входные данные – файл с расширением .py, оканчивающийся пустой строкой;
   * файл должен содержать исходный код из подмножества языка Python.
3. **Требования к выходным данным:**
   * выходные данные – ошибка, если один из анализаторов сообщил об ошибке, иначе файл с расширением .lua, содержащий код из подмножества языка Lua.
4. **Требования к пользователю программного средства:**
   * пользователь должен обладать начальными знаниями обращения с компьютером;
   * пользователь должен знать реализуемое подмножество входного языка (Python);
   * пользователь должен знать реализуемое подмножество выходного языка (Lua).

# 4 Синтаксис входного языка

G = (N, Σ, P, S),

N = {S, <программа>, <предложение>, <функция>, <идентификатор>, <символы идентификатора>, <идентификаторы>, <цифра>, <цифры>, <число>, <знак>, <вещественные>, <порядок>, <строка>, <кавычки строка>, <апостроф строка>, <символы>, <символ>, <юникод четверка>, <юникод пятерка>, <символ юникода>, <булеан>, <логическая операция>, <булевы выражения>, <выражения сравнения>, <математические выражения>, <операция сравнения>, <строковые выражения>, <первый приоритет>, <второй приоритет>, <третий приоритет>, <четвертый приоритет>, <объединенные выражения>, <левые выражения>, <вызов функции>, <коллекция>, <вывод>, <присваивание>, <операции эквивалентности>, <простое предложение>, <тело предложения>, <предложения>, <блок>, <условие>, <условное выражение>, <иначе>, <выражения>, <простые аргументы>, <названное выражение>, <именованные аргументы>, <аргументы>, <соответствие>, <соответствия>, <цикл>, <цикл while>, <цикл for>};

Σ = {**a**, **…** , **z**, **A**, **… Z**, **0**, **…** , **9**, **\_**, **(**, **{**, **[**, **)**, **}**, **]**, **.** , **,** , **+**, **-**, **"**, **'**, , **\U**, **!**, **\***, **%**, **\***, **/**, **=**, **<**, **>**, **\n**, **INDENT**, **DEDENT**, **:}**;

Схема грамматики P:

<S> → <программа>

<программа> → <предложение> | <функция> | <предложение><программа> | <функция><программа>

<идентификатор> → **A**<символы идентификатора> | … | **z**<символы идентификатора> | **\_**<символы идентификатора> | **A** | … | **z** | **\_**

<идентификаторы> → <идентификатор> | <идентификатор>**,** <идентификаторы>

<символы идентификатора> → **A**<символы идентификатора> | … | **z**<символы идентификатора> | **0**<символы идентификатора> | … | **9**<символы идентификатора> | **\_**<символы идентификатора> | **A** | … | **z** | **0** | … | **9** | **\_**

<цифра> → **0** | … | **9**

<цифры> → <цифра><цифры> | <цифра>

<число> → <знак><вещественные> | <знак><вещественные> <порядок> | <вещественные> | <вещественные> <порядок>

<вещественные> → <цифры>**.**<цифры>|**.**<цифры>|<цифры>**.**|<цифры>

<порядок> → **e**<знак><цифры> | **E**<знак><цифры> | **e**<цифры> | **E**<цифры>

<знак> → **+** | **-**

<строка> → **"**<кавычки строка> | **'**<апостроф строка>

<кавычки строка> → <символы>**"**

<апостроф строка> → <символы>**'**

<символы> → <символ><символы> | <символ>

<символ> → **\U0000**<юникод четверка> | **\U000**<юникод пятерка> | **0**<символ> | … | **9**<символ> | **A**<символ> | … | **z**<символ> | **\_**<символ>

<юникод четверка> → <символ юникода><символ юникода><символ юникода><символ юникода>

<юникод пятерка> → <символ юникода><символ юникода><символ юникода><символ юникода><символ юникода>

<символ юникода> → **0** | … | **9** | **A** | … | **F**

<булеан> → **True** | **False**

<логическая операция> → **and** | **or**

<булевы выражения> → <булеан> | **not** <булевы выражения> | <идентификатор> | <булевы выражения> <логическая операция> <булевы выражения> | <выражения сравнения>

<выражения сравнения> → <математические выражения> <операция сравнения> <математические выражения> | <строковые выражения> <операции эквивалентности><строковые выражения>

<математические выражения> → <первый приоритет>

<первый приоритет> → <второй приоритет> **+** <первый приоритет> | <второй приоритет> **-** <первый приоритет> | <второй приоритет>

<второй приоритет> → <третий приоритет> **\*** <второй приоритет> | <третий приоритет> **/** <второй приоритет> | <третий приоритет> **%** <второй приоритет> | <третий приоритет>

<третий приоритет> → <четвертый приоритет> **\*\*** <третий приоритет> | <четвертый приоритет>

<четвертый приоритет> → **(**<первый приоритет>**)** | <число> | <идентификатор> | <вызов функции> | <выражения длины>

<выражения длины> → **len(**<вызов функции>**)** | **len(**<идентификатор>**)** | **len(**<коллекция>**)**

<строковые выражения> → <строка> | <вызов функции> | <строка> **+** <строковые выражения> | <строковые выражения> **+** <идентификатор> | <идентификатор> **+** <строковые выражения> | <строковые выражения> **+** <идентификатор> **+** <строковые выражения>

<объединенные выражения> → <строковые выражения> | <математические выражения> | <булевы выражения> | <вызов функции> | <коллекция>

<левые выражения> → <строковые выражения> | <математические выражения> | <булевы выражения> | <вызов функции>

<вызов функции> → <идентификатор>**(**<аргументы>**) |** <идентификатор>**()**

<вывод> → **print(**<объединенные выражения>**)**

<присваивание> → <идентификатор> **=** <объединенные выражения>

<операция сравнения> → **<** | **<=** | **>** | **>=** | <операции эквивалентности>

<операции эквивалентности> → **!=** | **==**

<простое предложение> → <тело предложения>**\n | \n**

<сложное предложение> → <условие> | <цикл>

<предложение> → <простое предложение> | <сложное предложение>

<тело предложения> → <объединенные выражения> | <вывод> | <присваивание> | **pass** | **return** <объединенные выражения> | **len(**<вызов функции>**)**

<предложения> → <предложение><предложения> | <предложение>

<блок> → <простое предложение> | **\n INDENT** <предложения> **DEDENT**

<условие> → **if** <условное выражение>

<условное выражение> → <булевы выражения>**:** <блок><иначе> | <булевы выражения>**:** <блок>

<иначе> → **elif**<условное выражение> | **else:** <блок>

<выражения> → <объединенные выражения>**,** <выражения> | <объединенные выражения>

<простые аргументы> → <выражения>

<названное выражение> → <присваивание>**,** <названное выражение> | <присваивание>

<именованные аргументы> → <названное выражение>

<аргументы> → <простые аргументы> | <именованные аргументы>

<соответствие> → <левые выражения>**:** <объединенные выражения>

<соответствия> → <соответствие>**,** <соответствия> | <соответствие>

<коллекция> → **dict(**<именованные аргументы>**) | dict()** | **{**<аргументы>**}** | **{**<соответствия>**}** | **{}** | **[**<аргументы>**]** | **[]** | <идентификатор> | <вызов функции>

<цикл> → <цикл while> | <цикл for>

<цикл while> → **while** <булевы выражения>**:** <блок>

<цикл for> → **for** <идентификатор> **in** <коллекция>**:** <блок> | **for** <идентификатор> **in range** **(**<математические выражения>**):** <блок> | **for** <идентификатор> **in range** **(**<математические выражения>**,** <математические выражения>**):** <блок>

<функция> → **def** <идентификатор>**(**<идентификаторы>**):** <блок> | **def** <идентификатор>**():** <блок>

# 5 Контекстные условия

1. **Контекстные условия о правилах описания идентификаторов в программах [3]:**

* каждый из идентификаторов, используемых в программе, может быть описан один и более раз (для каждого идентификатора определяется его область действия, и описание должно относиться к этой области действия – модулю программы, подпрограммы, описание в формальных параметрах процедур и функций и т.д.).

1. **Контекстные условия о правилах использования идентификаторов в своей области действия [3]:**

* глобальные идентификаторы (областью действия в этом случае является вся программа), могут использоваться в любом месте программы;
* идентификаторы, описанные внутри функции, могут использоваться только в контексте функции.

1. **Контекстные условия, определяющие правила соответствия видов величин, входящих в синтаксические конструкции программ [3]:**

* количество формальных и фактических параметров функций должно совпадать;
* при присваивании тип переменной должен меняться в соответствии с фактической правой частью.

# 6 Проект системы

На рисунке 2 показан проект создаваемой системы – транслятора.

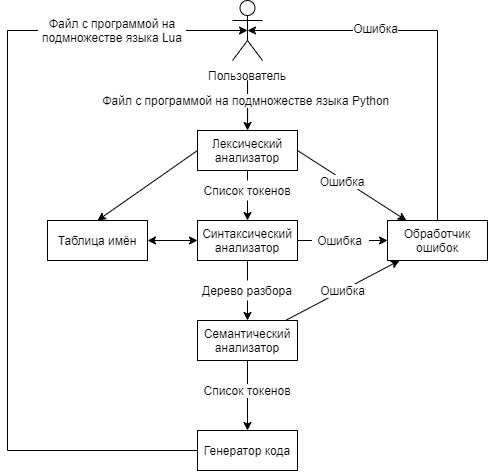


Рисунок 2 – Проект системы

На вход поступает файл с программой на подмножестве языка Python. После этого начинается работа лексического анализатора, который преобразует цепочку кода из входного файла в последовательность токенов. Эта последовательность передается синтаксическому анализатору, который создаёт дерево разбора. Далее семантический анализатор анализирует это дерево согласно контекстным условиям. Если при работе любого из анализаторов обнаруживаются ошибки, то пользователю выводятся эти ошибки. После работы всех анализаторов без ошибок будет сгенерирован файл с программой на подмножестве языка Lua.

# 7 Проект лексического анализатора

Лексема (лексическая единица языка) – это структурная единица языка, которая состоит из элементарных символов языка и не содержит в своём составе других структурных единиц языка [4].

Лексический анализатор – это часть транслятора, которая считывает исходную программу и выделяет в её тексте лексемы исходного языка. То есть результатом работы лексического анализатора является перечень всех найденных в тексте исходной программы лексем.

В ходе лексического анализа происходит перевод исходной программы на внутренний язык транслятора, в котором ключевые слова, идентификаторы, константы и т. д. приведены к одному формату. Одновременно с переводом исходной программы на внутренний язык на этапе лексического анализа проводится выявление в программе недопустимых слов [4].

Сгруппированные в лексемы входные символы преобразуются в токены, последовательность которых затем передаётся синтаксическому анализатору для дальнейшего разбора. Токен представляет собой четвёрку, состоящую из его типа (тип лексемы), значения, номера строки и позиции в строке (позиции в самого левого символа в строке).

## 7.1 Таблица ключевых слов

Ключевые слова – это предварительно определённые зарезервированные идентификаторы, имеющие специальные значения для транслятора.

В таблице 1 перечислены ключевые слова избранного подмножества языка Python, которые являются зарезервированными идентификаторами в любой части программы.

Таблица 1. Таблица ключевых слов

|  |  |
| --- | --- |
| **Ключевое слово** | **Ключ** |
| def | K1 |
| return | K2 |
| break | K3 |
| continue | K4 |
| pass | K5 |
| for | K6 |
| while | K7 |
| if | K8 |
| elif | K9 |
| else | K10 |
| print | K11 |
| len | K12 |
| dict | K13 |
| range | K14 |

## 7.2 Таблица зарезервированных имён

Зарезервированные имена – фиксированные имена переменных или функций, имеющие определенный смысл. Зарезервированные имена, как и ключевые слова, имеют специальные значения для транслятора, и потому не могут использоваться в качестве идентификаторов.

В таблице 2 перечислены слова, которые являются зарезервированными именами в избранном подмножестве языка Python.

Таблица 2. Таблица зарезервированных имен

|  |  |
| --- | --- |
| **Зарезервированное имя** | **Значение** |
| True | логическая константа (R1) |
| False | логическая константа (R2) |

## 7.3 Типы лексем

Тип лексемы – это код, который говорит о том, что данная лексема принадлежит одной из обозначенных нами групп.

Типы лексем избранного подмножества языка Python представлены в таблице 3.

Таблица 3. Типы лексем

|  |  |
| --- | --- |
| **Название типа** | **Описание типа** |
| Идентификатор | Начинающаяся с буквы латинского алфавита последовательность из символов латинского алфавита, цифр и \_ |
| Ключевое слово/зарезервированное имя | def, return, break, continue, pass, for, while, if, elif, else, print, len, dict, range, True, False |
| Оператор | + (O1)  - (O2)  \* (O3)  / (O4)  % (O5)  \*\* (O6)  == (O7)  != (O8)  < (O9)  > (O10)  <= (O11)  >= (O12)  not (O13)  and (O14)  or (O15)  in (O16)  = (O17) |
| Разделитель | , (D1)  ( (D2)  ) (D3)  { (D4)  } (D5)  [ (D6)  ] (D7)  INDENT (D8)  DEDENT (D9)  \n (D10)  : (D11) |
| Число | 1. Целое  2. Вещественное  3. В экспоненциальной форме |
| Строка | 1. Строка с апострофами (' ')  2. Строка с кавычками (" ") |

## 7.4 Представление лексического анализатора

Для проектирования лексического анализатора были использованы регулярные выражения, позволяющие легко распознавать введённые типы лексем и формировать соответствующие им токены.

Далее приведены регулярные выражения для языка Python, по шаблонам которых осуществляется поиск лексем в тексте входной программы до тех пор, пока не будет достигнут конец файла.

* Идентификатор

r"[A-Za-z\_][A-Za-z\_0-9]\*"

* Ключевое слово/зарезервированное имя

r"(?:\bdef\b|\breturn\b|\bbreak\b|\bcontinue\b|\bpass\b|\bfor\b|\bwhile\b|\bif\b|\belif\b|\belse\b|\bprint\b|\brange\b|\blen\b|\bin\b|\bdict\b|\bTrue\b|\bFalse\b)"

* Оператор

r"(?:\+|\-|\\*{1,2}|\/|\%|={1,2}|!=|<=|>=|<|>|\bnot\b|\band\b|\bor\b|\bin\b)"

* Разделитель:

r"(?:,|\(|\)|\[|\]|{|}|:)"

* Число

r"[+-]?(\d+([.]\d\*)?([eE][+-]?\d+)?|[.]\d+([eE][+-]?\d+)?)"

* Строка

r"(?:\".\*\"|\'.\*\')"

## 7.5 Ошибки лексического анализатора

Если в тексте исходной программы, будет встречен символ, не принадлежащий терминальному алфавиту, то анализатор выдаст лексическую ошибку, содержащую информацию о встреченном символе и его местонахождении (строка и номер символа в ней).

## 7.6 Работа лексического анализатора

Лексический анализатор получает на вход файл с исходным кодом и начинает посимвольно считывать программу. По мере возникновения совпадений при поиске по шаблону регулярного выражения осуществляется создание токенов, содержащих информацию о соответствующей лексеме.

Лексический анализатор работает, пока выполняются следующие условия:

* не достигнут конец файла;
* не произошла ошибка.

Схема работы лексического анализатора представлена на рисунке 3.

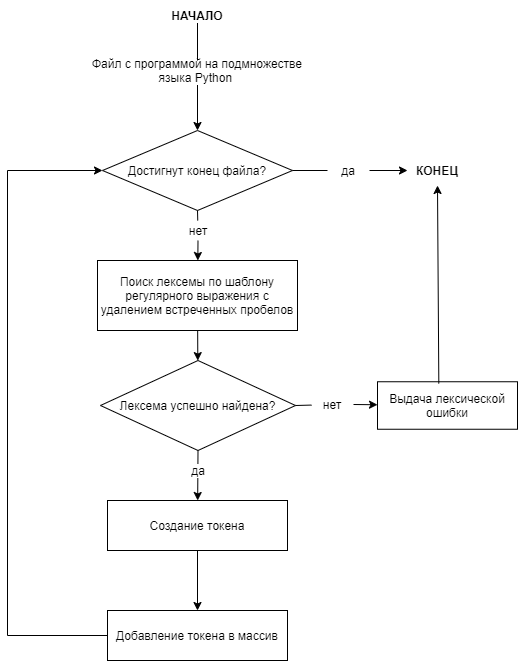


Рисунок 3 – Схема работы лексического анализатора

## 7.7 Модель данных

Лексический анализатор реализован классом **LexicalAnalyser**, имеющем функцию для получения последовательности токенов **parse(self, code)**, где code – текст программы на входном языке.

Для работы лексического анализатора также используются перечислимый класс **Type**, хранящий типы токенов; класс токена **Token**, имеющий конструктор **\_\_init\_\_(self, line: int, pos: int, content: str = "")**, функцию получения строкового представления **\_\_str\_\_(self)**, функцию получения содержания токена как нетерминала **as\_symbol(self)** и функцию инициализации типа токена **type(self)** и класс шаблона поиска AbstractPattern, имеющим функцию возвращения токена **token(self, match: str, line: int, pos: int)** и соответствующего для него регулярного выражения **regex(self)**.

Для определения различных типов токенов и регулярных выражений классы **Token** и **AbstractPattern** наследуются классами **NonTerminalToken**, **TokenIdentifier**, **TokenKeyword**, **TokenOperator**, **TokenDivider**, **TokenIndent**, **TokenNumber**, **TokenString** и **PatternNumber**, **PatternDivider**, **PatternOperator**, **PatternKeyword**, **PatternString**, **PatternIdentifier** соответственно.

Лексическая ошибка реализована в классе **LexicalError**, являющемся классом-потомком класса **AnalyzerError** ошибок транслятора.

Результат работы лексического анализатора представляет собой последовательность токенов (массив токенов), где токен – это структура с четырьмя полями:

* тип токена;
* значение;
* номер строки в исходной программе;
* позиция токена на этой строке (позиция самого левого символа токена).

# 8 Проект синтаксического анализатора

Задача синтаксического анализа – проверка правильности структуры программы, описываемой контекстно-свободной грамматикой.

Для синтаксического анализатора был выбран алгоритм Эрли.

## 8.1 Описание алгоритма

В дальнейшем символами *α*, *β*, и *γ* будем обозначать любую цепочку из символов объединенного алфавита (терминалов и/или нетерминалов) (включая пустую цепочку), *X* и *Y* обозначим отдельные нетерминалы, символом *a* – терминал.

Алгоритм Эрли представляет собой нисходящий алгоритм синтаксического анализа, применимый практически ко всем КС-грамматикам. Единственным ограничением на входную грамматику является отсутствие в ней циклов.

Алгоритм Эрли пытается построить все возможные нетерминалы из подстрок входной строки. Читая входную строку символ за символом, алгоритм для каждой позиции входной строки формирует список всех тех частично завершенных правил грамматики, из которых прочитанный префикс входной строки и его части могут быть выведены и после чтения очередного символа этот список модифицируется [5].

В работе алгоритма используется метасимвол точки: для правила *X* → *αβ*, запись *X* → *α* • *β* представляет случай, в котором *α* уже была распознана и ожидается обработка *β*.

Состояние рабочей среды алгоритма есть множество ситуаций. На каждом шаге алгоритма строится очередное состояние рабочей среды. Последовательность этих состояний называется списком разбора для входной цепочки. Нумерация состояний начинается с 0. Список разбора является выходом алгоритма.

Каждое состояние содержит список объектов, называемых ситуациями. Ситуация представляет собой кортеж [*X* → *α* • *β*, *i*], состоящий из:

1. рассматриваемого правила *X* → *α* *β*, согласно которому в данный момент считывается сегмент входной цепочки, выводящийся в соответствии с этим правилом;
2. места в правиле, показывающего, какая доля правой части этого правила уже распознана (обозначается точкой);
3. указателя *i* позиции во входной цепочке, после которого начался поиск возможности применения этого правила [5].

Анализатор начинает свою работу с состояния 0 с добавления в него ситуаций вида [*S* → • *α*, 0], где *S* — аксиома грамматики, затем повторно выполняя, пока это возможно, три вида операций: предсказание, считывание и завершение.

* Предсказание: для каждой ситуации в состоянии *Ik* вида [*X* → *α* • *Y β*, *j*] добавить в состояние *Ik* ситуации [*Y* → • *γ*, *k*] для каждого правила грамматики с *Y* в левой части правила *Y* → *γ*.
* Сканирование: если *a* является следующим символом входной цепочки, для каждой ситуации в *Ik* вида [*X* → *α* • *a β*, *j*] добавить в *Ik+1* ситуации [*X* → *α a* • *β*, *j*].
* Завершение: для каждой ситуации в *Ik* вида [*Y* → *γ* •, *j*] найти все ситуации в *Ij* вида [*X* → *α* • *Y β*, *i*] и добавить ситуации [*X* → *α Y* • *β*, *i*] в *Ik*.

Повторы ситуаций в состоянии не допускаются: новые ситуации добавляются лишь в том случае, если такая ситуация не содержится в данном состоянии. Вышеперечисленные три операции выполняются, пока не станет невозможным добавление новых ситуаций в состояние.

Считается, что алгоритм корректно завершил работу и цепочка принадлежит языку, порождаемому грамматикой, если ситуация вида [*S* → *γ* •, 0] содержится в *In*, где *S* — аксиома грамматики, *n* – длина цепочки, а также общее количество полученных состояний равно *n* + 1. В противном случае входная цепочка содержит ошибку.

## 8.2 Модель данных

Алгоритм Эрли реализован классом EarleyParse.

Для работы необходимо проинициализировать объект этого класса: задать формальную грамматику для разбора класса Grammar и массив токенов класса Token, полученных в результате работы лексического анализатора.

Формальная грамматика представлена в виде класса Grammar.

**сlass Grammar:**

**def add(self, rule)** // метод добавляет правило (**class Rule**) в грамматику;

**def load\_grammar(file)** // статический метод, загружающий грамматику из файла, путь к которому указан. Возвращает инициализированную грамматику;

**def \_\_getitem\_\_(self, nt)** // возвращает все правила, в правых частях которых находится указанный нетерминал;

**def is\_terminal(self, sym)** // проверяет, является ли данный символ терминальным;

**def is\_tag(self, sym)** // проверяет, является ли символ нетерминалом, который переводится правилами только в терминалы.

Табличный список разбора алгоритма Эрли представлен в виде класса Chart.

**class Chart:**

**def init(l)** // статический метод, инициализирует таблицу с *l* состояниями.

Состояние в таблице представлено в виде класса ChartEntry.

**class ChartEntry:**

**def add(self, state)** // добавляет ситуацию в состояние.

Ситуация алгоритма Эрли представлена в виде класса EarleyState.

**class EarleyState:**

**def next(self)** // возвращает символ, стоящий в правиле после точки;

**def is\_complete(self)** // проверяет, завершено ли состояние, то есть находится ли точка в конце правой части правила;

**def init()** // статический метод, возвращающий начальное состояние, из которого программа может перейти к аксиоме заданной грамматики.

Алгоритм разбора реализован в классе EarleyParse.

**class EarleyParse:**

**def predictor(self, state, pos)** // добавляет ситуации, в левой части которых находится нетерминал, стоящий в данном состоянии после точки;

**def scanner(self, state, pos)** // добавляет в следующее состояние ситуацию, в которой точка перенесена через терминал, находящийся на соответствующей позиции во входной цепочке;

**def completer(self, state, pos) //** добавляет в это состояние ситуацию из состояния с указанным номером, в которой точка стоит перед нетерминалом, который стоит в левой части поданной на вход ситуации и переносит точку через нетерминал;

**def parse(self)** // выполняет алгоритм Эрли для данной цепочки;

**def \_get(self)** // строит дерево разбора цепочки.

Для описания дерева разбора используется класс **Tree** библиотеки nltk.tree.

Синтаксическая ошибка реализована в классе **SyntacticError**, являющемся классом-потомком класса **AnalyzerError** ошибок транслятора.

## 8.3 Полный текст грамматики в компьютерно-читаемом виде

S -> <program>

<program> -> <sentence> | <function> | <sentence> <program> | <function> <program>

<Identifiers> -> <Identifier> | <Identifier> , <Identifiers>

<boolean> -> True | False

<logical\_operation> -> and | or | <equivalence\_operations>

<boolean\_expressions> -> <boolean> | not <boolean\_expressions> | <Identifier> | <boolean\_expressions> <logical\_operation> <boolean\_expressions> | <comparison\_expressions>

<comparison\_expressions> -> <mathematical\_expressions> <comparison\_operations> <mathematical\_expressions> | <String\_expressions> <equivalence\_operations> <String\_expressions>

<mathematical\_expressions> -> <first\_priority>

<first\_priority> -> <second\_priority> + <first\_priority> | <second\_priority> - <first\_priority> | <second\_priority>

<second\_priority> -> <third\_priority> \* <second\_priority> | <third\_priority> / <second\_priority> | <third\_priority> % <second\_priority> | <third\_priority>

<third\_priority> -> <fourth\_priority> \*\* <third\_priority> | <fourth\_priority>

<fourth\_priority> -> ( <first\_priority> ) | <Number> | <Identifier> | <function\_call> | <length\_expressions>

<length\_expressions> -> len ( <function\_call> ) | len ( <Identifier> ) | len ( <collection> )

<String\_expressions> -> <String> | <function\_call> | <String> + <String\_expressions> | <String\_expressions> + <Identifier> | <Identifier> + <String\_expressions> | <String\_expressions> + <Identifier> + <String\_expressions>

<concatenated\_expressions> -> <Identifier> | <String\_expressions> | <mathematical\_expressions> | <boolean\_expressions> | <function\_call> | <collection>

<left\_expressions> -> <Identifier> | <String\_expressions> | <mathematical\_expressions> | <boolean\_expressions> | <function\_call>

<function\_call> -> <Identifier> ( <arguments> ) | <Identifier> ( )

<output> -> print ( <concatenated\_expressions> )

<assignment> -> <Identifier> = <concatenated\_expressions>

<comparison\_operations> -> < | <= | > | >= | <equivalence\_operations>

<equivalence\_operations> -> != | ==

<simple\_sentence> -> <sentence\_body> newline | newline

<complex\_sentence> -> <condition> | <cycle>

<sentence> -> <simple\_sentence> | <complex\_sentence>

<sentence\_body> -> <concatenated\_expressions> | <output> | <assignment> | pass | return <concatenated\_expressions> | len ( <function\_call> )

<sentences> -> <sentence> <sentences> | <sentence>

<block> -> <simple\_sentence> | newline indent <sentences> dedent

<condition> -> if <conditional>

<conditional> -> <boolean\_expressions> : <block> <otherwise> | <boolean\_expressions> : <block>

<otherwise> -> elif <conditional> | else : <block>

<expressions> -> <concatenated\_expressions> , <expressions> | <concatenated\_expressions>

<arguments> -> <expressions>

<named\_expression> -> <assignment> , <named\_expression> | <assignment>

<named\_arguments> -> <named\_expression>

<match> -> <left\_expressions> : <concatenated\_expressions>

<matches> -> <match> , <matches> | <match>

<collection> -> dict ( <named\_arguments> ) | dict ( ) | { <arguments> } | { <matches> } | { } | [ <arguments> ] | [ ] | <Identifier> | <function\_call>

<cycle> -> <cycle> | <for\_loop>

<cycle> -> while <boolean\_expressions> : <block>

<for\_loop> -> for <Identifier> in <collection> : <block> | for <Identifier> in range ( <mathematical\_expressions> ) : <block> | for <Identifier> in range ( <mathematical\_expressions> , <mathematical\_expressions> ) : <block>

<function> -> def <Identifier> ( <Identifiers> ) : <block> | def <Identifier> ( ) : <block>

## 8.4 Проект алгоритма

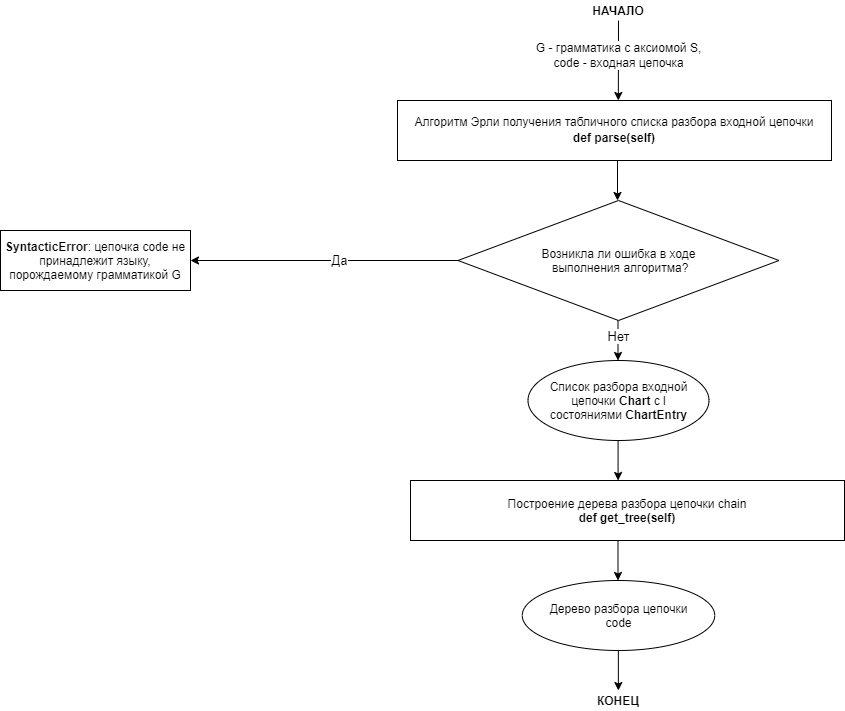


Рисунок 4 – Блок-схема работы синтаксического анализатора

## 8.5 Работа синтаксического анализатора

Синтаксический анализатор получает на вход последовательность лексем (входная цепочка). Затем выполняется алгоритм Эрли получения списка разбора для цепочки, после чего, в случае, если разбор цепочки в данной грамматике возможно найти, строится дерево разбора цепочки.

Входная цепочка является допустимой, если в состоянии *In* содержится ситуация вида [*S* → *γ* •, 0], где *S* – аксиома грамматики, *n* – длина цепочки, а также общее количество полученных состояний равно *n* + 1.

Если ошибок не возникло, то строится дерево разбора.

Пусть входная цепочка имеет вид:

def meaning\_of\_life(x):

return 42

variable = 21

print(meaning\_of\_life(variable))

Дерево разбора данной входной цепочки приведено на рисунке 5.

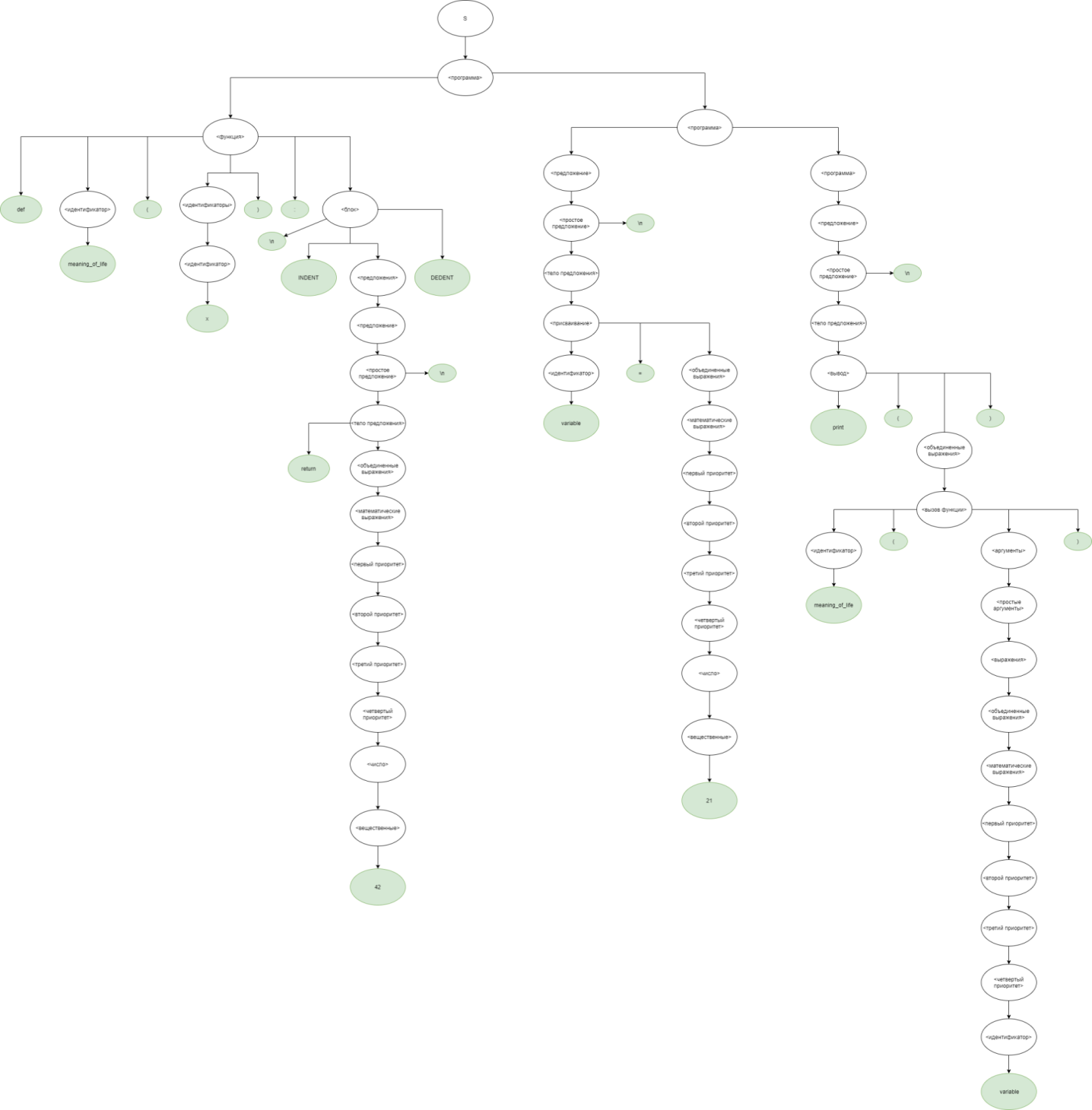


Рисунок 5 – Дерево разбора входной цепочки

## 8.6 Ошибки синтаксического анализатора

Процесс синтаксического анализа в нашем случае обнаруживает ошибку, если цепочка не принадлежит языку, порождаемому грамматикой. В таком случае анализатор выдаст синтаксическую ошибку, содержащую информацию о невозможности выполнения синтаксического анализа с использованием алгоритма Эрли.

# 9 Проект семантического анализатора

Семантический анализатор использует синтаксическое дерево для проверки исходной программы на семантическую согласованность с определением языка [2]. В ходе семантического анализа проверяется смысл программы, иными словами, проверяются контекстные условия языка.

## 9.1 Работа семантического анализатора

Семантический анализатор получает на вход дерево разбора, полученное после работы синтаксического анализатора. При обходе дерева, если в узле встречается использование идентификатора:

1. Если это присваивание, то в таблицу текущего контекста переменной добавляется название переменной.

2. Если это описание функции, то выполняется подсчёт количества формальных параметров функции, и в таблице контекстов создаётся новая таблица контекста функции.

3. Если это вызов функции, то выполняется проверка соответствия количества фактических параметров функции и формального описания, а также проверка всех переменных, которые необходимо захватить из внешнего контекста.

4. Иначе это использование идентификатора и выполняется проверка наличия его объявления.

В случае возникновения семантической ошибки анализатор заканчивает работу с выдачей этой ошибки.

## 9.2 Ошибки семантического анализатора

Если идентификатор переменной, используемый в программе, не был объявлен, то ошибка будет выглядеть следующим образом:

*Ошибка 1: Идентификатор переменной был встречен до её объявления (The identifier was encountered before it was announced)*

Если используется идентификатор функции, но при этом сама функция не была описана, то ошибка будет выглядеть следующим образом:

*Ошибка 2: Функция была вызвана до её объявления (The function identifier was used before it was announced)*

Если количество формальных и фактических параметров функции не совпадает, то ошибка будет выглядеть следующим образом:

*Ошибка 3: Количество формальных и фактических параметров функции не совпадает (Parameters in the declaration and function call do not match).*

Если идентификатор переменной используется внутри контекста вызываемой функции, но при этом переменная не была объявлена в этом контексте и не является глобальной, то ошибка будет выглядеть следующим образом:

*Ошибка 4: В контексте вызываемой функции встречена неизвестная переменная (When the function was called, the variable used in it was not declared)*

Если функция вызывается внутри контекста другой, вызываемой функции, но при этом первая не была описана в этом контексте, то ошибка будет выглядеть следующим образом:

*Ошибка 5: В контексте вызываемой функции встречена неизвестная функция (When the function was called, the function name used in it was not declared)*

# 10 Проект генератора кода

Генерация объектного кода является заключительным этапом работы транслятора, в ходе которого происходит перевод на выходной язык программы, проверенной на синтаксическую и семантическую корректность.

На рисунке 6 изображён проект генератора кода.

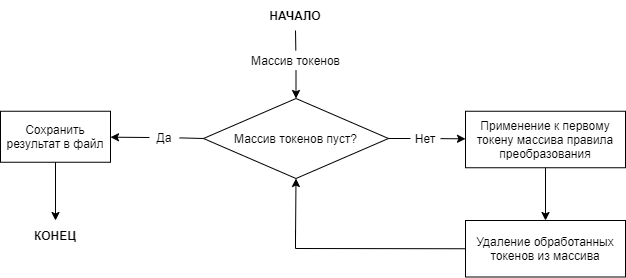


Рисунок 6 – Проект генератора кода

## 10.1 Модель данных

Генератор кода реализован функцией **generate(tokens)**, принимающей на вход массив токенов tokens и возвращающей результат генерации в файл на выходном языке.

Также используются вспомогательные функции **generate\_table(dict\_tokens: List[Token])** и **as\_proxy\_list(tokens)** для обработки коллекций и цикла for.

## 10.2 Таблица соответствия

При помощи таблицы соответствия (таблица 4) отображаются элементы языка Python, эквивалентные элементам языка Lua. В ней отображаются как представление отдельных лексем, так и структура программы, процедур и функций.

Таблица 4. Соответствие Python – Lua

|  |  |
| --- | --- |
| **Python** | **Lua** |
| 1. Структура программы | |
| <программа> | <программа> |
| 2. Циклы | |
| 2.1 for <идентификатор> in <коллекция>:  <предложения>  2.2 while <булево выражение>:  <предложения> | 2.1 for <идентификатор> in pairs(<коллекция>) do  <предложения>  end  2.2 while <булево выражение> do  <предложения>  end |
| 3. Условие | |
| if <булево выражение>:  <предложения>  elif <булево выражение>:  <предложения>  else:  <предложения> | if <булево выражение> then <предложения>  end  elseif <булево выражение> then <предложения>  end  else  <предложения>  end |
| 4. Логические константы | |
| True  False | true  false |
| 5. Логические операции | |
| not  and  or | not  and  or |
| 6. Математические операции | |
| +  -  \*  /  %  \*\* | +  -  \*  /  %  ^ |
| 7. Функция | |
| def <идентификатор> (<формальные параметры>):  <предложения> | function <идентификатор> (<формальные параметры>)  <предложения>  end |
| 8. Операции сравнения | |
| ==  !=  <  >  <=  >= | ==  ~=  <  >  <=  >= |

# 11 Тестирование

## 11.1 Тестирование лексического анализатора

Тестовая ситуация 1

Описание ситуации: программа без лексических ошибок

Входные данные:

print("Hello!")

Ожидаемый результат:

Status: SUCCESS

Результат: Тест пройден.

Тестовая ситуация 2

Описание ситуации: программа с недопустимым символом

Входные данные:

#print("Hello!")

Ожидаемый результат:

Status: ERROR

Description: Lexical Error

Incorrect code in position 1 line 1: #print("Hello!")

↑

Результат: Тест пройден.

## 11.2 Тестирование синтаксического анализатора

Тестовая ситуация 1

Описание ситуации: пустая программа

Входные данные: пустой файл с расширением .py

Ожидаемый результат: пустой файл с расширением .lua

Результат: Тест пройден.

Тестовая ситуация 2

Описание ситуации: программа без синтаксических ошибок

Входные данные:

a = 2

if a > 0:

print('Ok')

Ожидаемый результат:

Status: SUCCESS

Результат: Тест пройден.

Тестовая ситуация 3

Описание ситуации: программа с синтаксической ошибкой в условии

Входные данные:

a = 2

if a > 0

print('Ok')

Ожидаемый результат:

Status: ERROR

Description: Syntactic Error

Unable to parse code using Earley Parser

Результат: Тест пройден.

Тестовая ситуация 4

Описание ситуации: программа с синтаксической ошибкой в цикле

Входные данные:

for i range(2):

print(i)

Ожидаемый результат:

Status: ERROR

Description: Syntactic Error

Unable to parse code using Earley Parser

Результат: Тест пройден.

Тестовая ситуация 5

Описание ситуации: программа с пустой функцией

Входные данные:

a = 1

print(a)

def foo():

pass

Ожидаемый результат:

Status: SUCCESS

Результат: Тест пройден.

Тестовая ситуация 6

Описание ситуации: программа с функцией с несколькими аргументами

Входные данные:

a = 1

print(a)

def foo():

pass

def baz(f, c, m):

print(f)

baz(a, a, a)

Ожидаемый результат:

Status: SUCCESS

Результат: Тест пройден.

## 11.3 Тестирование семантического анализатора

Тестовая ситуация 1

Описание ситуации: ошибка «идентификатор переменной встречен до её объявления»

Входные данные:

print(a)

Ожидаемый результат:

Status: ERROR

Description: Semantic Error

The identifier was encountered before it was announced:

Token:

token\_type = Type.Identifier

content = a

line = 1

position = 7

Результат: Тест пройден.

Тестовая ситуация 2

Описание ситуации: ошибка «функция вызвана до её описания»

Входные данные:

foo()

Ожидаемый результат:

Status: ERROR

Description: Semantic Error

The function identifier was used before it was announced:

Token:

token\_type = Type.Identifier

content = foo

line = 1

position = 1

Результат: Тест пройден.

Тестовая ситуация 3

Описание ситуации: ошибка «количество формальных и фактических параметров функции не совпадает»

Входные данные:

def foo(a, b):

return a + b

foo(2)

Ожидаемый результат:

Status: ERROR

Description: Semantic Error

Parameters in the declaration and function call do not match:

Token:

token\_type = Type.Identifier

content = foo

line = 3

position = 1

Результат: Тест пройден.

Тестовая ситуация 4

Описание ситуации: ошибка «в контексте вызываемой функции встречена неизвестная переменная»

Входные данные:

def foo(a, b):

a = c + 2

return a + b

foo(3, 2)

Ожидаемый результат:

Status: ERROR

Description: Semantic Error

When the function was called, the variable used in it was not declared:

Token:

token\_type = Type.Identifier

content = foo

line = 4

position = 1

Результат: Тест пройден.

Тестовая ситуация 5

Описание ситуации: ошибка «в контексте вызываемой функции встречена неизвестная функция»

Входные данные:

def foo(a, b):

a = foo2() + 2

return a + b

foo(3, 2)

Ожидаемый результат:

Status: ERROR

Description: Semantic Error

When the function was called, the function name used in it was not declared:

Token:

token\_type = Type.Identifier

content = foo

line = 4

position = 1

Результат: Тест пройден.

## 11.4 Тестирование генератора кода

Тестовая ситуация 1

Описание ситуации: генерация циклов

Входные данные:

a = 0

while a < 10:

print('iteration')

a = a + 1

Ожидаемый результат:

a = 0

while a < 10 do

print('iteration')

a = a + 1

end

Результат: Тест пройден.

Тестовая ситуация 2

Описание ситуации: генерация условий

Входные данные:

b = 16

if b == 16:

b = 2

elif b == 2:

b = 1

else:

b = 0

Ожидаемый результат:

b = 16

if b == 16 then

b = 2

elseif b == 2 then

b = 1

else

b = 0

end

Результат: Тест пройден.

Тестовая ситуация 3

Описание ситуации: генерация логических операций

Входные данные:

a = True

b = False

c = a or b

Ожидаемый результат:

a = true

b = false

c = a or b

Результат: Тест пройден.

Тестовая ситуация 4

Описание ситуации: генерация операторов

Входные данные:

a = 1

if a != 3:

if a < 3:

a = 3

else:

a = 2

Ожидаемый результат:

a = 1

if a ~= 3 then

if a < 3 then

a = 3

else

a = 2

end

end

Результат: Тест пройден.

Тестовая ситуация 5

Описание ситуации: генерация математических операций

Входные данные:

a = 5 \*\* 5 % 5 + 5 - 5 \* 5 / 10

Ожидаемый результат:

a = 5 ^ 5 % 5 + 5 - 5 \* 5 / 10

Результат: Тест пройден.

Тестовая ситуация 6

Описание ситуации: генерация функций

Входные данные:

def func(a, b, c):

return a + b \* c

print(func(1, 2, 3))

Ожидаемый результат:

function func(a, b, c)

return a + b \* c

end

print(func(1, 2, 3))

Результат: Тест пройден.

# Заключение

В рамках работы были выполнены поставленные задачи:

* разработана грамматика подмножества языка Python;
* описаны контекстные условия входного языка;
* описано соответствие конструкций входного и выходного языков;
* разработан проект лексического анализатора;
* разработан проект синтаксического анализатора;
* разработан проект семантического анализатора;
* разработан проект генератора кода.
* разработанные проекты реализованы в виде программы-транслятора.

# Список литературы

1. Ахо, Альфред В., Лам, Моника С., Сети, Рави, Ульман, Джеффри Д. Компиляторы: принципы, технологии и инструментарий, 2-е изд.: Пер. с англ. – М.: ООО «И.Д. Вильямс», 2008. – 1184 с.: ил. – Парал. тит. англ.
2. Артемьева, И. Л. Теория вычислительных процессов и структур. Часть 1. Языки и способы их задания. Учебно-методическое пособие. Владивосток: ДВФУ, 2011, 60 с.
3. Братчиков, И. Л. Синтаксис языков программирования / И. Л. Братчиков. – М.: Наука, 1975. – 232 с.
4. Молчанов, А. Ю. Системное программное обеспечение: лабораторный практикум / Молчанов Алексей Юрьевич. - Санкт-Петербург [и др.]: Питер, 2005. - 283 с.: ил. - (Учебное пособие)
5. Нисходящий алгоритм Эрли // Файловый архив для студентов – StudFiles. [2018]. Дата обновления: 30.11.2018. URL: <https://studfile.net/preview/7300347/page:4/> (дата обращения 22.06.2021)