Министерство науки и высшего образования Российской Федерации

Федеральное государственное бюджетное образовательное учреждение

высшего образования

**«КУБАНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»**

**(ФГБОУ ВО «КубГУ»)**

**Факультет компьютерных технологий и прикладной математики**

**Кафедра вычислительных технологий**

**Отчёт**

**по лабораторной работе №1**

**Дисциплина: Методы разработки трансляторов**

**Тема: «Построение лексического анализатора»**

Работу выполнил \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ А. М. Нагалевский

Направление подготовки 02.03.02 Фундаментальная информатика и

информационные технологии

Направленность (профиль) Математическое и программное обеспечение

компьютерных технологий

Преподаватель

д-р техн. наук, проф. \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Ю. М. Вишняков

**Содержание**

[1 Вариант задания 3](#_Toc97893076)

[2 Основные понятия лексического анализа 3](#_Toc97893077)

[3 Функции и таблицы лексического анализа 4](#_Toc97893078)

[4 Диаграмма состояний лексического процессора 9](#_Toc97893079)

[5 Результаты экспериментов 15](#_Toc97893080)

[Приложение А Описание правил записи элементов заданного входного языка 20](#_Toc97893081)

[Приложение Б Листинг программы и комментарии к нему 24](#_Toc97893082)

**1 Вариант задания**

Вариант задания представляет собой пару: входной язык и выходной язык (таблица 1).

Таблица 1 – Вариант задания

|  |  |  |
| --- | --- | --- |
| № варианта | Входной язык | Выходной язык |
| 42 | Java | C++ |

Разработать лексический анализатор, корректно распознающий все лексемы и формирующего таблицы лексем и внутреннее представление проанализированного текста.

На вход программы подается файл, содержащий текст на входном языке программирования. Результатом работы программы должен быть файл, содержащий последовательность кодов лексем входной программы, а также один или несколько файлов, содержащих все таблицы лексем.

Отчет по работе должен содержать описание правил записи элементов заданного входного языка, стандартные таблицы лексем, диаграмма состояний сканера, листинг программы и комментарии к нему, пример лексического разбора.

**2 Основные понятия лексического анализа**

Лексический анализ всегда предшествует синтаксическому анализу и заключается в предварительной обработке программы и ее перекодировании к виду, удобному для синтаксического анализа. Лексический анализ принято отделять от синтаксического анализа для сокращения времени компиляции, поскольку синтаксис (морфология) слов всегда проще, чем синтаксис программ.

Лексический анализ сводится к разбиению текста программы на лексические единицы (слова, лексемы) – конструкции, которые выступают в качестве терминальных символов для синтаксического анализа. Лексемы еще иногда называют символами или атомами.

Большинство лексем в языках программирования можно сгруппировать в следующие классы:

1. класс идентификаторов;
2. класс служебных слов;
3. класс констант (числовых или символьных);
4. класс операций (одно-, дву- или многолитерных);
5. класс разделителей (однолитерных или двулитерных).

В лексическом анализе лексема представляется в виде пары (класс, значение).

**3 Функции и таблицы лексического анализа**

Основная функция лексического анализатора (сканера) состоит в выделении лексем из исходной программы и передаче их синтаксическому анализатору в некотором внутреннем представлении. При этом лексический анализатор использует таблицы, соответствующие классам лексем. Таблицы служебных слов, разделителей и операций определяются входным языком и должны быть сформированы при построении сканера, они являются постоянными и неизменяемыми. Таблицы идентификаторов и констант являются временными и создаются непосредственно в процессе лексического разбора исходной программы.

Комментарии в процессе лексического анализа игнорируются, так как они специфицируют исходную программу только для программиста и не влияют на семантику программы.

Для рассматриваемого языка сформированные таблицы служебных слов (таблица 2), таблица операций (таблица 3) и таблица разделителей (таблица 4) имеют вид:

Таблица 2 – Таблица служебных слов

|  |  |
| --- | --- |
| Служебное слово | Код |
| abstract | 1 |
| case | 2 |
| continue | 3 |
| extends | 4 |
| goto | 5 |
| int | 6 |
| package | 7 |
| short | 8 |
| try | 9 |
| assert | 10 |
| catch | 11 |
| default | 12 |
| final | 13 |
| if | 14 |
| private | 15 |
| static | 16 |
| this | 17 |
| void | 18 |
| boolean | 19 |
| сhar | 20 |
| do | 21 |

Продолжение таблицы 2

|  |  |
| --- | --- |
| Служебное слово | Код |
| long | 22 |
| protected | 23 |
| throw | 24 |
| volatile | 25 |
| break | 26 |
| class | 27 |
| double | 28 |
| float | 29 |
| import | 30 |
| native | 31 |
| public | 32 |
| super | 33 |
| throws | 34 |
| while | 35 |
| byte | 36 |
| const | 37 |
| else | 38 |
| for | 39 |
| instanceof | 40 |
| new | 41 |
| return | 42 |
| switch | 43 |
| transient | 44 |
| print | 45 |
| println | 46 |
| main | 47 |
| System | 48 |
| out | 49 |
| String | 50 |
| args | 51 |

Таблица 3 – Таблица операций

|  |  |
| --- | --- |
| Операция | Код |
| \* | 1 |
| + | 2 |
| - | 3 |
| % | 4 |
| / | 5 |
| ++ | 6 |
| \*= | 7 |
| =+ | 8 |
| =- | 9 |
| =% | 10 |
| =/ | 11 |
| == | 12 |
| < | 13 |
| <= | 14 |
| != | 15 |
| ! | 16 |
| > | 17 |
| >= | 18 |
| & | 19 |
| | | 20 |

Таблица 4 – Таблица разделителей

|  |  |
| --- | --- |
| Разделитель | Код |
| табуляция | 1 |
| перевод строки | 2 |
| пробел | 3 |
| ( | 4 |
| ) | 5 |
| , | 6 |
| . | 7 |
| : | 8 |
| ; | 9 |
| [ | 10 |
| ] | 11 |
| { | 12 |
| } | 13 |

Каждая лексема, обработанная сканером, преобразуется к виду:

<буква><код>,

где: <буква> – это признак класса лексемы (W, I, O, R, N или C),   
<код> – номер лексемы в соответствующей таблице.

При лексическом разборе входной цепочки пробелы не кодируются как разделители и в выходную цепочку не попадают. Это связано с тем, что в исходном тексте программы пробелы служат только для структурирования текста программы и не несут дополнительной смысловой нагрузки.

**4 Диаграмма состояний лексического процессора**

Лексический анализатор представляет собой процессор или, иначе говоря, сканер (конечный автомат) для разбора и классификации лексем, связанный с некоторыми семантическими процедурами. На вход такого процессора последовательно подаются символы исходной программы, причем каждый входной символ вызывает изменение состояния сканера.

Если анализируемый символ означает конец разбираемой лексемы, то с переходом связывается некоторая семантическая процедура, позволяющая либо определить код лексемы по таблице (в случае служебных слов, разделителей, операций), либо пополнить таблицы (для констант и идентификаторов), а затем в выходную строку выдать очередной код лексемы во внутреннем представлении. Для эффективного разбора и оптимизации сканера, а также для его программной реализации в алфавите автомата выделяют несколько подмножеств символов, по которым сканер выполняет одинаковые переходы и действия. В нашем случае такими подмножествами являются:

1. буквы {A, …, Z, a, …, z};
2. цифры {0, …, 9};
3. разделители {табуляция, перевод строки, пробел, «(», «)», «,», «.», «:», «;», «[», «]»,«{», «}»};
4. операции {\*, +, -,%, /,=\*,=+,=-,=%,=/,==, <, <=, !=, =, >, >=,&,|};
5. точка;
6. минус;
7. плюс;
8. одинарная кавычка;
9. слэш;
10. фигурные скобки;
11. конец строки
12. конец файла.

С учетом вышесказанного диаграмма состояний сканера для рассматриваемого языка имеет вид, представленный на рисунках 1-7.

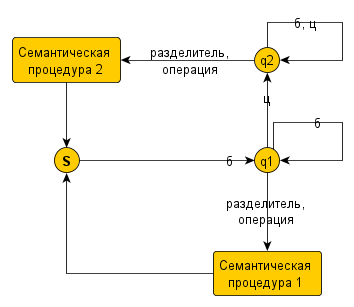


Рисунок 1 – Диаграмма состояний сканера для распознавания служебного слова или идентификатора

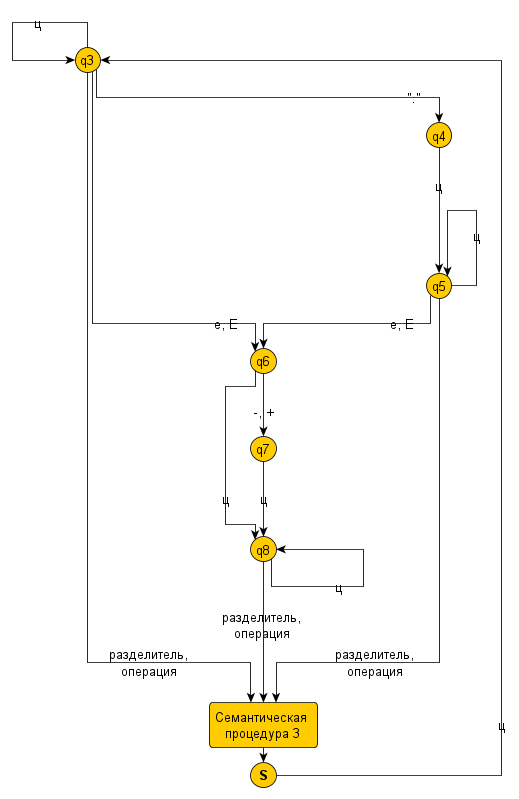


Рисунок 2 – Диаграмма состояний сканера для распознавания числовой константы

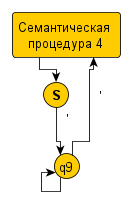


Рисунок 3 – Диаграмма состояний сканера для распознавания символьной константы

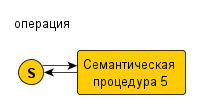


Рисунок 4 – Диаграмма состояний сканера для распознавания операции

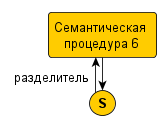


Рисунок 5 – Диаграмма состояний сканера для распознавания разделителя

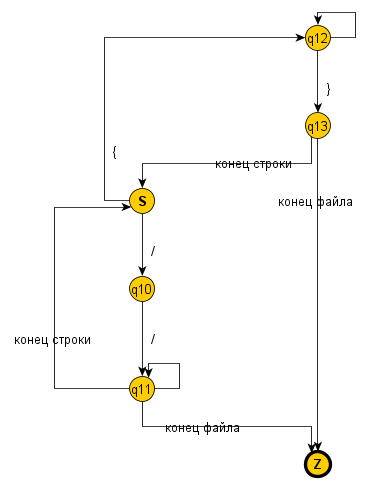


Рисунок 6 – Диаграмма состояний сканера для распознавания комментария

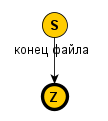


Рисунок 7 – Диаграмма состояний сканера для определения конца файла

На данной диаграмме состояния сканера представлены вершинами, а переходы между состояниями – дугами (направленными линиями). Каждый переход связан с чтением очередного символа их текста входной программы. Поэтому дуга взвешена (помечена) символом или множеством символов, которые вызывают данный переход. Если в диаграмме состояний есть невзвешенная дуга, ведущая из какого-либо состояния, то считается, что она взвешена любыми символами кроме тех, которыми взвешены другие дуги, исходящие из данного состояния. Это позволяет не перегружать диаграмму лишними символами. Скругленный прямоугольник в разрыве дуги указывает на семантическую процедуру, выполняемую при данном переходе. Если переход не сопровождается семантической процедурой, то текущий символ добавляется к буферу, в котором формируется лексема. Работа каждой семантической процедуры завершается очищением буфера.

Сканер всегда содержит три стандартных состояния:

1. S – начальное состояние сканера. Лексический разбор всегда начинается из этого состояния.
2. Z – заключительное состояние сканера. Если в процессе разбора достигнуто данное состояние, это означает, что разбор успешно завершен.
3. F – состояние ошибки. Если встретился символ, не входящий во входной алфавит, то сканер переходит в состояние ошибки и прекращает разбор. Кроме того, если в каком-либо состоянии на входе появился символ, по которому не предусмотрен переход из этого состояния, сканер также переходит в состояние ошибки (на диаграмме эти переходы не изображаются, чтобы избежать лишнего загромождения рисунка).

За семантическими процедурами процессора закреплены следующие функции:

Семантическая процедура 1: Провести поиск сформированного слова в таблицах служебных слов, операций. Если такое служебное слово в таблице служебных слов, операция в таблице операций не найдены, то выполнить Семантическую процедуру 2, иначе сформировать и выдать в выходную последовательность лексему служебного слова или операции во внутреннем представлении.

Семантическая процедура 2: Провести поиск сформированного слова в таблице идентификаторов. Если такое слово в таблице идентификаторов не найдено, то занести сформированное слово в таблицу идентификаторов. Сформировать и выдать в выходную последовательность лексему идентификатора во внутреннем представлении.

Семантическая процедура 3: Занести сформированное слово в таблицу числовых констант. Сформировать и выдать в выходную последовательность лексему числовой константы во внутреннем представлении.

Семантическая процедура 4: Занести сформированное слово в таблицу символьных констант. Сформировать и выдать в выходную последовательность лексему символьной константы во внутреннем представлении.

Семантическая процедура 5: Провести поиск сформированного слова в таблице операций. Сформировать и выдать в выходную последовательность лексему операции во внутреннем представлении.

Семантическая процедура 6: Провести поиск текущего символа в таблице разделителей. Сформировать и выдать в выходную последовательность лексему разделителя во внутреннем представлении.

**5 Результаты экспериментов**

Пример работы программы для программы для тестирования: на рисунке 8 представлен скриншот файла, содержащего текст на входном языке программирования, на рисунке 9 – скриншот файла, содержащего последовательность кодов лексем входной программы, на рисунках   
10-15 – скриншоты файлов, содержащие все лексемы.

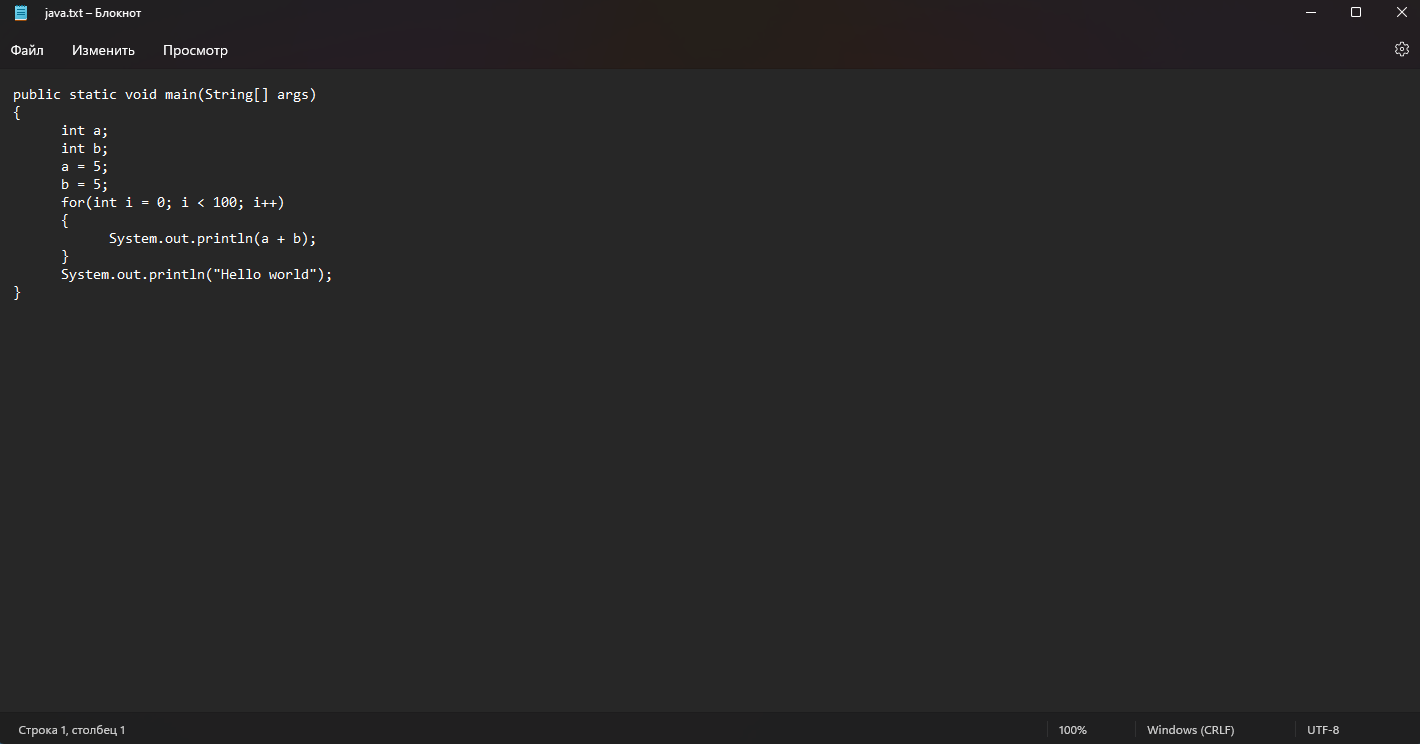


Рисунок 8 – Скриншот файла, содержащего текст на входном языке программирования

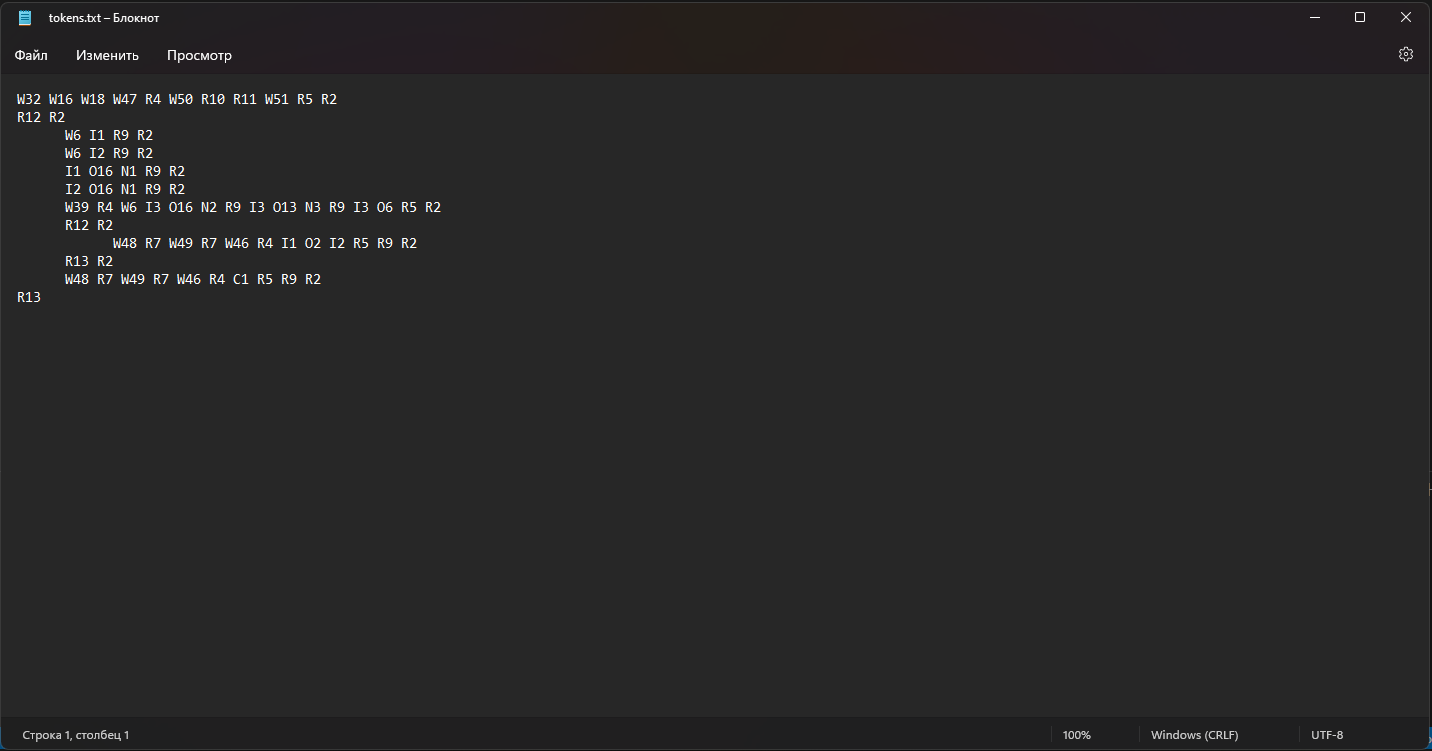


Рисунок 9 – Скриншот файла, содержащего последовательность кодов лексем входной программы

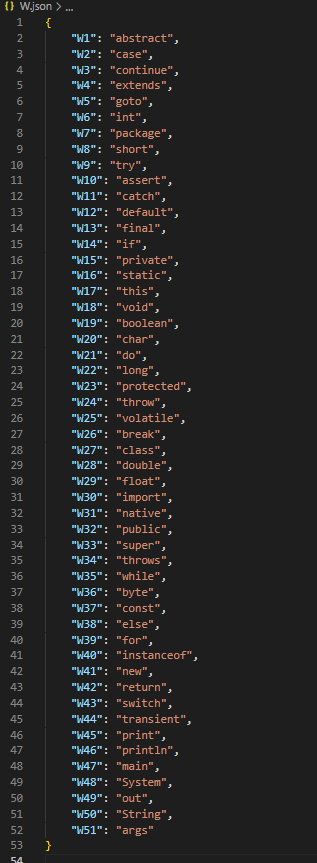


Рисунок 10 – Скриншот файла, содержащего лексемы служебных слов

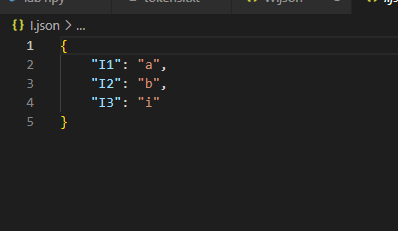


Рисунок 11 – Скриншот файла, содержащего лексемы идентификаторов



Рисунок 12 – Скриншот файла, содержащего лексемы числовых констант

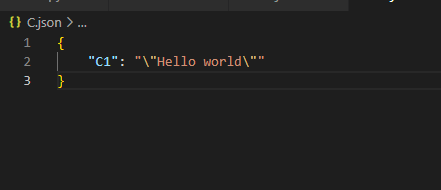


Рисунок 13 – Скриншот файла, содержащего лексемы символьных констант

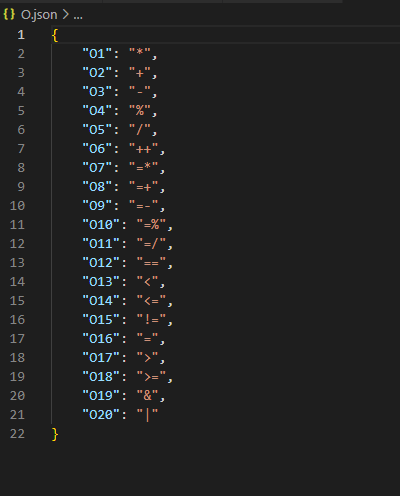


Рисунок 14 – Скриншот файла, содержащего лексемы операций



Рисунок 15 – Скриншот файла, содержащего лексемы разделителей

**Приложение А   
Описание правил записи элементов заданного входного языка**

***Язык программирования Java***

**Идентификаторы**

Произвольная последовательность букв и цифр, начинающаяся с буквы. Может включать символы подчеркивания.

**Числовые константы целого типа**

Произвольная последовательность цифр без знака.

**Числовые константы вещественного типа, представленные с фиксированной точкой**

Последовательность цифр, включающая одну десятичную точку вида

123.45

**Символьные (строковые) константы**

Последовательность символов, заключенная в кавычки, расположенная в пределах одной строки, вида:

“Hello world 123 %\_ крот”

**Переменные с индексами (массивы и элементы массивов)**

Тип массива, квадратные скобки и идентификатор, после которого через запятую перечислены выражения‑индексы, вида:

int[] abc = {1,2,3,4}

abc[1+i]

**Комментарии (строчные и блочные)**

Блочные – последовательность символов, заключенная в /\* \*/, возможно содержащая несколько строк:

/\* Это комментарий,

Который содержит 2 строки \*/

Строчные – от символов «//» до конца строки.

i:=i+1; // это инкремент

**Обращения к процедурам и функциям пользователя**

Идентификатор, после которого в круглых скобках следует последовательность выражений‑аргументов, разделенных запятыми. Отсутствие аргументов допускается:

f(12, 4, i)

f()

**Арифметические операции**

Сложение +

Вычитание -

Умножение \*

Деление /

Целочисленное деление div

Остаток от деления mod

**Операции сравнения**

Меньше <

Больше >

Равно =

Не равно <>

Меньше или равно <=

Больше или равно >=

**Оператор присваивания**

Имеет вид «=». Слева стоит идентификатор, а справа – выражение. Заканчивается символом «;», например:

a=b+с;

**Операторы блока**

static/public { – начало блока

…

} - конец блока

**Оператор описания программы**

Программа начинается оператором public class с указанием имени программы. Затем могут идти описания данных, процедур и функций, а затем тело программы, заключенное в операторы блока, оканчивающееся точкой.

public class <идентификатор> {

…

}

**Операторы описания данных (идентификаторов и массивов)**

Начинается со слова типа и перечисления идентификаторов через запятую.

int a,c;

float b=100.2;

Типы переменных: int (целый), float (вещественный), string (строковый)

**Операторы описания процедур и функций**

Процедуры имеют заголовок вида

public/private <идентификатор> (<список формальных параметров>){

и тело – список операторов, заключенный в операторы блока

}

В остальном структура функций аналогична структуре процедур.

**Оператор безусловного перехода и метки**

goto <метка>;

Метка - идентификатор, расположенный в теле программы в начале строки, после которого стоит знак «:»:

label:

print(“Hi”);

**Оператор условного перехода**

Начинается с ключевого слова «if», имеет полный и неполный формат:

if <условие> then <оператор\_1> else <оператор\_2>;

if <условие> then <оператор\_1>;

**Арифметические циклы**

Синтаксис:

for (переменная = значение 1; значение 1 оператор сравнения значение 2; переменная оператор);

Оператор for вызывает оператор по одному разу для каждого значения в диапазоне от значения 1 до значения 2.

**Итерационные циклы с предусловием**

Синтаксис:

while (Логическое выражение) {

оператор;

}

Оператор после { будет выполняться до тех пор, пока логическое выражение принимает истинное значение (True). Логическое выражение является условием возобновления цикла. Его истинность проверяется каждый раз перед очередным повторением оператора цикла, который будет выполняться лишь до тех пор, пока логическое выражение истинно. Как только логическое выражение принимает значение ложь (False), осуществляется переход к оператору, следующему за while.

Выражение оценивается до выполнения оператора, так что если оно с самого начала было ложным (False), то оператор не будет выполнен ни разу.

**Итерационные циклы с постусловием**

Синтаксис:

do {

оператор;

оператор

...

Оператор;

} while (выражение);

Операторы между словами do и while повторяются, пока логическое выражение является ложным (False). Как только логическое выражение становится истинным (True), происходит выход из цикла.

Так как выражение оценивается после выполнения операторов, то в любом случае операторы выполнятся хотя бы один раз.

**Операторы завершения цикла**

Для всех операторов цикла выход из цикла осуществляется как вследствие естественного окончания оператора цикла, так и с помощью операторов перехода и выхода.

Break

Continue

Return

Процедура Break выполняет безусловный выход из цикла. Процедура Continue обеспечивает переход к началу новой итерации цикла. Процедуры Return передает программное управление обратно в вызывающую программу.

**Приложение Б   
Листинг программы и комментарии к нему**

**import json**

**from tkinter import \***

**SERVICE\_WORDS = ['abstract', 'case', 'continue', 'extends', 'goto', 'int', 'package', 'short', \**

**'try', 'assert', 'catch', 'default', 'final', 'if', 'private', \**

**'static', 'this', 'void', 'boolean', 'char', 'do','long', 'protected', \**

**'throw', 'volatile', 'break', 'class', 'double', 'float', 'import', 'native', \**

**'public', 'super','throws','while','byte','const','else','for','instanceof',\**

**'new','return','switch','transient','print','println','main','System',\**

**'out','String','args']**

**OPERATIONS = ['\*','+','-','%', '/','++','=\*','=+','=-','=%','=/','==', '<', '<=', '!=', '=', '>', '>=','&','|']**

**SEPARATORS = ['\t', '\n', ' ', '(', ')', ',', '.', ':', ';', '[', ']','{','}']**

**def check(tokens, token\_class, token\_value):**

**if not(token\_value in tokens[token\_class]):**

**token\_code = str(len(tokens[token\_class]) + 1)**

**tokens[token\_class][token\_value] = token\_class + token\_code**

**def get\_operation(input\_sequence, i):**

**for k in range(2, 0, -1):**

**if i + k < len(input\_sequence):**

**buffer = input\_sequence[i:i + k]**

**if buffer in OPERATIONS:**

**return buffer**

**return ''**

**def get\_separator(input\_sequence, i):**

**buffer = input\_sequence[i]**

**if buffer in SEPARATORS:**

**return buffer**

**return ''**

**tokens = {'W': {}, 'I': {}, 'O': {}, 'R': {}, 'N': {}, 'C': {}}**

**for service\_word in SERVICE\_WORDS:**

**check(tokens, 'W', service\_word)**

**for operation in OPERATIONS:**

**check(tokens, 'O', operation)**

**for separator in SEPARATORS:**

**check(tokens, 'R', separator)**

**def prog():**

**f = open('java2.txt', 'r')**

**input\_sequence = f.read()**

**f.close()**

**i = 0**

**state = 'S'**

**output\_sequence = buffer = ''**

**while i < len(input\_sequence):**

**symbol = input\_sequence[i]**

**operation = get\_operation(input\_sequence, i)**

**separator = get\_separator(input\_sequence, i)**

**if state == 'S':**

**buffer = ''**

**if symbol.isalpha():**

**state = 'q1'**

**buffer += symbol**

**elif symbol.isdigit():**

**state = 'q3'**

**buffer += symbol**

**elif symbol == "'":**

**state = 'q9'**

**buffer += symbol**

**elif symbol == '"':**

**state = 'q10'**

**buffer += symbol**

**elif symbol == '/':**

**state = 'q11'**

**elif operation:**

**check(tokens, 'O', operation)**

**output\_sequence += tokens['O'][operation] + ' '**

**i += len(operation) - 1**

**elif separator:**

**if separator != ' ':**

**check(tokens, 'R', separator)**

**output\_sequence += tokens['R'][separator]**

**if separator == '\n':**

**output\_sequence += '\n'**

**else:**

**output\_sequence += ' '**

**elif i == len(input\_sequence) - 1:**

**state = 'Z'**

**elif state == 'q1':**

**if symbol.isalpha():**

**buffer += symbol**

**elif symbol.isdigit():**

**state = 'q2'**

**buffer += symbol**

**else:**

**if operation or separator:**

**if buffer in SERVICE\_WORDS:**

**output\_sequence += tokens['W'][buffer] + ' '**

**elif buffer in OPERATIONS:**

**output\_sequence += tokens['O'][buffer] + ' '**

**else:**

**check(tokens, 'I', buffer)**

**output\_sequence += tokens['I'][buffer] + ' '**

**if operation:**

**check(tokens, 'O', operation)**

**output\_sequence += tokens['O'][operation] + ' '**

**i += len(operation) - 1**

**if separator:**

**if separator != ' ':**

**check(tokens, 'R', separator)**

**output\_sequence += tokens['R'][separator]**

**if separator == '\n':**

**output\_sequence += '\n'**

**else:**

**output\_sequence += ' '**

**state = 'S'**

**elif state == 'q2':**

**if symbol.isalnum():**

**buffer += symbol**

**else:**

**if operation or separator:**

**check(tokens, 'I', buffer)**

**output\_sequence += tokens['I'][buffer] + ' '**

**if operation:**

**check(tokens, 'O', operation)**

**output\_sequence += tokens['O'][operation] + ' '**

**i += len(operation) - 1**

**if separator:**

**if separator != ' ':**

**check(tokens, 'R', separator)**

**output\_sequence += tokens['R'][separator]**

**if separator == '\n':**

**output\_sequence += '\n'**

**else:**

**output\_sequence += ' '**

**state = 'S'**

**elif state == 'q3':**

**if symbol.isdigit():**

**buffer += symbol**

**elif symbol == '.':**

**state = 'q4'**

**buffer += symbol**

**elif symbol == 'e' or symbol == 'E':**

**state = 'q6'**

**buffer += symbol**

**else:**

**if operation or separator:**

**check(tokens, 'N', buffer)**

**output\_sequence += tokens['N'][buffer] + ' '**

**if operation:**

**check(tokens, 'O', operation)**

**output\_sequence += tokens['O'][operation] + ' '**

**i += len(operation) - 1**

**if separator:**

**if separator != ' ':**

**check(tokens, 'R', separator)**

**output\_sequence += tokens['R'][separator]**

**if separator == '\n':**

**output\_sequence += '\n'**

**else:**

**output\_sequence += ' '**

**state = 'S'**

**elif state == 'q4':**

**if symbol.isdigit():**

**state = 'q5'**

**buffer += symbol**

**elif state == 'q5':**

**if symbol.isdigit():**

**buffer += symbol**

**elif symbol == 'e' or symbol == 'E':**

**state = 'q6'**

**buffer += symbol**

**else:**

**if operation or separator:**

**check(tokens, 'N', buffer)**

**output\_sequence += tokens['N'][buffer] + ' '**

**if operation:**

**check(tokens, 'O', operation)**

**output\_sequence += tokens['O'][operation] + ' '**

**i += len(operation) - 1**

**if separator:**

**if separator != ' ':**

**check(tokens, 'R', separator)**

**output\_sequence += tokens['R'][separator]**

**if separator == '\n':**

**output\_sequence += '\n'**

**else:**

**output\_sequence += ' '**

**state = 'S'**

**elif state == 'q6':**

**if symbol == '-' or symbol == '+':**

**state = 'q7'**

**buffer += symbol**

**elif symbol.isdigit():**

**state = 'q8'**

**buffer += symbol**

**elif state == 'q7':**

**if symbol.isdigit():**

**state = 'q8'**

**buffer += symbol**

**elif state == 'q8':**

**if symbol.isdigit():**

**buffer += symbol**

**else:**

**if operation or separator:**

**check(tokens, 'N', buffer)**

**output\_sequence += tokens['N'][buffer] + ' '**

**if operation:**

**check(tokens, 'O', operation)**

**output\_sequence += tokens['O'][operation] + ' '**

**i += len(operation) - 1**

**if separator:**

**if separator != ' ':**

**check(tokens, 'R', separator)**

**output\_sequence += tokens['R'][separator]**

**if separator == '\n':**

**output\_sequence += '\n'**

**else:**

**output\_sequence += ' '**

**state = 'S'**

**elif state == 'q9':**

**if symbol != "'":**

**buffer += symbol**

**elif symbol == "'":**

**buffer += symbol**

**check(tokens, 'C', buffer)**

**output\_sequence += tokens['C'][buffer] + ' '**

**state = 'S'**

**elif state == 'q10':**

**if symbol != '"':**

**buffer += symbol**

**elif symbol == '"':**

**buffer += symbol**

**check(tokens, 'C', buffer)**

**output\_sequence += tokens['C'][buffer] + ' '**

**state = 'S'**

**elif state == 'q11':**

**if symbol == '/':**

**state = 'q12'**

**elif symbol == '\*':**

**state = 'q13'**

**elif state == 'q12':**

**if symbol == '\n':**

**state = 'S'**

**elif i == len(input\_sequence) - 1:**

**state = 'Z'**

**elif state == 'q13':**

**if symbol == '\*':**

**state = 'q14'**

**elif state == 'q14':**

**if symbol == '/':**

**state = 'q15'**

**elif state == 'q15':**

**if symbol == '\n':**

**state = 'S'**

**elif i == len(input\_sequence) - 1:**

**state = 'Z'**

**i += 1**

**f2 = open('tokens.txt', 'w')**

**new\_output\_sequence = output\_sequence.replace('R1 ','\t')**

**f2.write(new\_output\_sequence)**

**f2.close()**

**def tokenizzz():**

**for token\_class in tokens.keys():**

**with open('%s.json' % token\_class, 'w') as write\_file:**

**data = {val: key for key, val in tokens[token\_class].items()}**

**json.dump(data, write\_file, indent=4, ensure\_ascii=False)**

**def write\_txt(data):**

**with open('java2.txt','w') as file:**

**file.write(data)**

**def clicked():**

**write\_txt(codetxt.get("1.0","end"))**

**prog()**

**tokenizzz()**

**f3=open('java.txt','r')**

**beginLabel=Label(window,text=f3.read(),font=("Arial",13))**

**beginLabel.place(x=40,y=300)**

**f3.close()**

**f4=open('tokens.txt','r')**

**endLabel=Label(window,text=f4.read(),font=("Arial",13))**

**endLabel.place(x=400,y=300)**

**f4.close()**

**window=Tk()**

**window.title("LR1")**

**window.geometry('900x600')**

**codetxt=Text(window)**

**codetxt.place(x=40,y=0,width=410,height=250)**

**btngo=Button(window,text="Выполнить",command=clicked)**

**btngo.place(x=470,y=0,width=100,height=50)**

**window.mainloop()**