ГУАП

КАФЕДРА № 44

ОТЧЕТ
ЗАЩИЩЕН С ОЦЕНКОЙ

ПРЕПОДАВАТЕЛЬ

канд. техн. наук, доцент		Н.В. Кучин
должность, уч. степень, звание	подпись, дата	инициалы, фамилия

ОТЧЕТ О ЛАБОРАТОРНОЙ РАБОТЕ №1

ПОСТРОЕНИЕ РАСПОЗНАВАТЕЛЯ ДЛЯ РЕГУЛЯРНОЙ ГРАММАТИКИ И ЛЕКСИЧЕСКОГО АНАЛИЗАТОРА

по курсу: Системное программное обеспечение

РАБОТУ ВЫПОЛНИЛ

СТУДЕНТ гр. № 4143		Д. В. Пономарев
	подпись, дата	инициалы, фамилия

Санкт-Петербург 2023

Вариант №14

1. Цель работы: Изучение основных понятий теории регулярных языков и грамматик, ознакомление с назначением и принципами работы конечных автоматов (КА) и лексических анализаторов (сканеров). Получение практических навыков построения КА на основе заданной регулярной грамматики. Получение практических навыков построения сканера на примере заданного простейшего входного языка.

2. Задание:

Построить регулярную грамматику в соответствии с вариантом задания.

Вариант 14. Входной язык содержит логические выражения, разделенные символом ; (точка с запятой). Логические выражения состоят из идентификаторов, шестнадцатеричных чисел, знака присваивания (:=), знаков операций or, xor, and, not и круглых скобок.

Написать программу, которая выполняет лексический анализ входного текста в соответствии с заданием и порождает таблицу лексем с указанием их типов и значений. Текст на входном языке задаётся в виде символьного (текстового) файла. Программа должна выдавать сообщения о наличие во входном тексте ошибок, которые могут быть обнаружены на этапе лексического анализа. Наличие синтаксических ошибок проверять не требуется.

Длину идентификаторов и строковых констант можно считать ограниченной 32 символами.

Любые лексемы, не предусмотренные вариантом задания, встречающиеся в исходном тексте, должны трактоваться как ошибочные.

Также при открытии скобок они всегда должны быть закрыты, в случае когда находиться лишняя скобка она обозначается как ошибка.

3. Описание регулярной грамматики (с помощью регулярных выражений).

Далее представлены описания лексем в виде регулярных выражений.

- Знак присваивания: [:=]
- Конец выражения (знак): [;]
- Идентификатор: [a-zA-Z][a-zA-Z0-9] (буква, за которой следует любое количество букв или цифр)
- Число (в шестнадцатеричной системе счисления): [0-9A-F]+ (одна или более цифр от 0 до 9 или буквы от A до F)
- Знаки операций: or|xor|and|not (это вариант "или" в регулярном выражении, что означает, что одно из этих слов должно совпадать)

4. Граф переходов КА для распознавания лексем.

На рисунке 1 граф переходов для распознавания лексем.

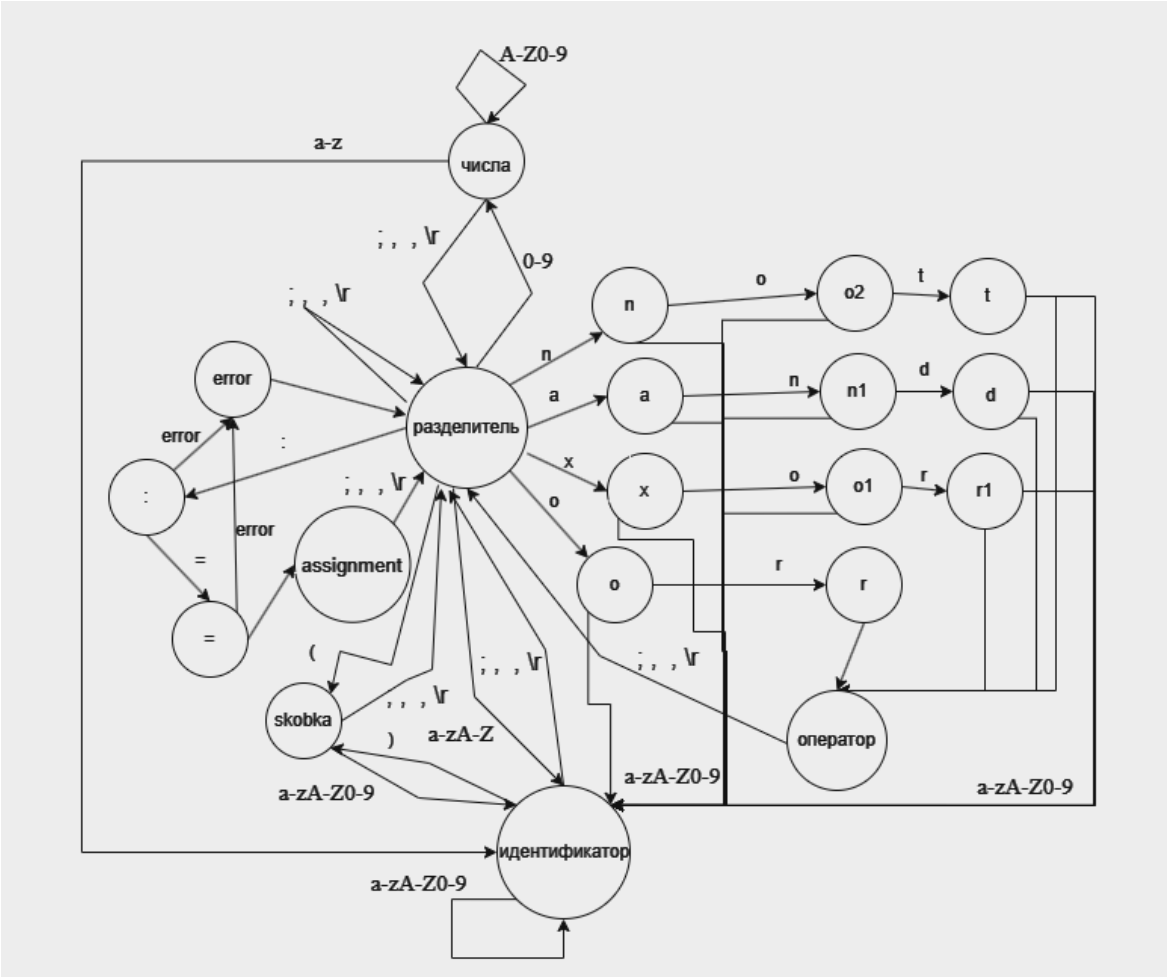

Рисунок 1 — Граф переходов для распознавания лексем.

5. Пример анализируемого входного текста и результат работы лексического анализатора

Описание типов лексем:

Идентификатор – del

Шестнадцатеричные числа – number

Знака присваивания – assignment

Знаки операций **or**, **xor**, **and**, **not** – operator

Круглые скобоки – skobka

Текст анализируемой программы с ошибками

z:= y or not(x));

x:= 2 + 3;

(index)	type	lexem
0	'identifier'	'z'
1	'assignment'	':='
2	'identifier'	'y'
3	'operator'	'or'
4	'operator'	'not'
5	'skobka'	'('
6	'identifier'	'x'
7	'skobka'	')'
8	'error'	')'
9	'del'	';'
10	'identifier'	'x'
11	'assignment'	':='
12	'number'	'2'
13	'error'	'+'
14	'number'	'3'
15	'del'	';'

Рисунок 2 – Результат работы программы с ошибками

На рисунке 2 видно две ошибки. Первая это лишняя скобка, вторая это недопустимый символ “+”.

Текст анализируемой программы без ошибками

c:=2 xor (3 or 2);

y:= x or not y;

x:= 2E and 3FC;

(index)	type	lexem
0	'identifier'	'c'
1	'assignment'	':='
2	'number'	'2'
3	'operator'	'xor'
4	'skobka'	'('
5	'number'	'3'
6	'operator'	'or'
7	'number'	'2'
8	'skobka'	')'
9	'del'	';'
10	'identifier'	'y'
11	'assignment'	':='
12	'identifier'	'x'
13	'operator'	'or'
14	'operator'	'not'
15	'identifier'	'y'
16	'del'	';'
17	'identifier'	'x'
18	'assignment'	':='
19	'number'	'2E'
20	'operator'	'and'
21	'number'	'3FC'
22	'del'	';'

Рисунок 3 — Результат обработки файла без ошибок

6. Программа

Программа написана на языке JavaScript. Она считывает текст из файла, разбивает его на символы и затем эмулирует работу конечного автомата для распознавания лексем. Она использует набор состояний, определяя текущее состояние и в зависимости от этого состояния выбирает следующий шаг для анализа символов.

Основные состояния включают состояния для разделителей, чисел, идентификаторов и операторов. Каждое состояние имеет свой набор правил для обработки символов. Программа также использует вспомогательные

функции для обработки различных сценариев, таких как обработка исключений и завершение лексем, включая операторы.

В процессе работы программа создает список лексем с соответствующими типами (разделитель, число, оператор, идентификатор или ошибка), который затем выводится для анализа.

Текст программы

```
const { error } = require('console');
const fs = require('fs');

// Путь к файлу, который нужно прочитать
const filePath = 'test.txt';

// Наборы знаков с помощью которых можно определить следующее состояние
const Alldels = [';', ' ', '\r','\n'];
const Alphabet =
'qwertyuiopasdfghjklzxcvbnmQWERTYUIOPASDFGHJKLZXCVBNM1234567890'.split('');
const numbers = '1234567890'.split('');
const Hexnumbers = '1234567890ABCDEF'.split('');
// Вспомогательный класс для хранения и отображения лексем
class lexemSaver {
  constructor() {
    // Задать массив лексем
    this.lexems = [];
  }
  // Добавить лексему
  add(state, lexem) {
    this.lexems.push({ type: typeMap[state] || 'identifier', lexem: lexem });
  }
  // Напечатать сохраненные лексемы
  print() {
    console.log('Лексемы:');
    console.table(this.lexems);
  }
}
// Карта типов для лексем завершающихся в каком либо из состояний
const typeMap = {
  'del': 'del',
  'number': 'number',
  'r': 'operator',
  'r1': 'operator',
  'd': 'operator',
  't': 'operator',
  ':=': 'assignment',
  '(': 'skobka',
  ')': 'skobka',
  'error': 'error',
  ':': 'error',
};



// Основная функция выступающая в качестве лексера
async function lexer() {
```

```
  // Инициализируем начальные состояния
  let state = 'del';// Начальное состояние - разделитель
  let lexem = '';// Текущая лексема
  let skobch = 0;// Счетчик открытых скобок
  const lexems = new lexemSaver();
// Получаем символы из файла
  const charact = await getcharact(filePath);

  // Итерируемся по каждому символу
  for (let i = 0; i < charact.length; i++) {
    // Свитч по состояниям программы
    switch (state) {
      case 'del': { // Состояние прохода по разделяющим лексемы символам
        if (lexem) lexems.add(state, lexem);// Добавляем лексему в
сохраненные, если она не пустая
        lexem = charact[i].trim();// Обрезаем пробелы у текущего символа
        switch (true) {
          case Alldels.includes(charact[i]): state = 'del'; break;// Если
текущий символ - разделитель, остаемся в состоянии del
          case numbers.includes(charact[i]): state = 'number'; break;// Если
текущий символ - цифра, переходим в состояние number
          case 'o' == charact[i]: state = 'o'; break;//переходим на обработку
or чтобы понять идентификатор это или оператор
          case 'x' == charact[i]: state = 'x'; break;//переходим на обработку
xor чтобы понять идентификатор это или оператор
          case 'a' == charact[i]: state = 'a'; break;//переходим на обработку
and чтобы понять идентификатор это или оператор
          case 'n' == charact[i]: state = 'n'; break;//переходим на обработку
not чтобы понять идентификатор это или оператор
          case '(' == charact[i]: state = '('; break;// переходимна
обработчики скобок
          case ')' == charact[i]: state = ')'; break;
          case Alphabet.includes(charact[i]): state = 'identifier'; break;
          case ':' == charact[i]: state = ':'; break;// переходим на
обработку знаков присваивания
          default: {
            lexems.add('error', charact[i]);
            lexem = '';
            state = 'del';
          } break;
        }
      } break;
      //обратываем оператор or
      case 'o': ({ state, lexem } = VariantExclude(lexems, lexem, state,
charact[i], 'r', 'r')); break;
      case 'r': ({ state, lexem } = VariantDefault(lexems, lexem, state,
charact[i])); break;
//обратываем оператор xor
      case 'x': ({ state, lexem } = VariantExclude(lexems, lexem, state,
charact[i], 'o', 'o1')); break;
      case 'o1': ({ state, lexem } = VariantExclude(lexems, lexem, state,
charact[i], 'r', 'r1')); break;
      case 'r1': ({ state, lexem } = VariantDefault(lexems, lexem, state,
charact[i])); break;
//обратываем оператор and
      case 'a': ({ state, lexem } = VariantExclude(lexems, lexem, state,
charact[i], 'n', 'n1')); break;
      case 'n1': ({ state, lexem } = VariantExclude(lexems, lexem, state,
charact[i], 'd', 'd')); break;
      case 'd': ({ state, lexem } = VariantDefault(lexems, lexem, state,
charact[i])); break;
//обратываем оператор not
      case 'n': ({ state, lexem } = VariantExclude(lexems, lexem, state,
charact[i], 'o', 'o2')); break;
```

```
      case 'o2': ({ state, lexem } = VariantExclude(lexems, lexem, state,
charact[i], 't', 't')); break;
      case 't': ({ state, lexem } = VariantDefault(lexems, lexem, state,
charact[i])); break;

      case ':=': ({ state, lexem } = VariantOperatorEnd(lexems, lexem, state,
charact[i])); break;

      case 'identifier': { // Состояние поиска идентификаторов
        switch (true) {
          case Alphabet.includes(charact[i]): { // Продолжение текущей
лексемы
            lexem += charact[i];
            state = 'identifier';
          } break;
          default: ({ state, lexem } = VariantDefault(lexems, lexem, state,
charact[i]));
        }
      } break;
      case 'operator': { // Состояние поиска операторов
        switch (true) {
          case Alldels.includes(charact[i]): { // Переход к следующей лексеме
            lexems.add(state, lexem);
            lexem = charact[i];
            state = 'del';
          } break;
          case Alphabet.includes(charact[i]): { // Продолжение текущей
лексемы
            lexem += charact[i];
            state = 'identifier';
          } break;
          default: { // Выдача ошибки в случае неправильного ввода
            lexems.add('error', lexem + charact[i]);
            lexem = '';
            state = 'del';
          }
        }
      } break;
      case 'number': { // Состояния поиска чисел
        switch (true) {
          case Alldels.includes(charact[i]): { // Переход к следующей лексеме
            lexems.add(state, lexem);
            lexem = charact[i].trim();
            state = 'del';
          } break;
          case Hexnumbers.includes(charact[i]): { // Продолжение поиска числа
            lexem += charact[i];
            state = 'number';
          } break;
          case ')'== charact[i]: {
            lexems.add(state, lexem);
            lexem = charact[i];
            state = ')';
          } break;
          default: { // Выдача ошибки в случае неправильного ввода
            lexems.add('error', lexem + charact[i]);
            lexem = '';
            state = 'del';
          }
        }
      } break;

      // проверяем чтобы существовал символ присваивания
      case ':': {
```

```
        switch (true) {
          case '=' == charact[i]: { lexem += charact[i]; state = ':='; };
break;// после : всегда должен быть = иначе ошибка
          default: {
            lexems.add('error', lexem + charact[i]);
            lexem = '';
            state = 'del';
          }
        }
      } break;
      // обрабатываеми скобки, так как если скобка была открыта она должна
быть всегда закрыта, для этого добавим счетчик
      case '(': {
        skobch +=1;// прибавляем 1 открытую скобку
         ({ state, lexem } = VariantOperatorEnd(lexems, lexem, state,
charact[i]));
      } break;
      case ')': {
        if (skobch > 0)// проверяем есть ли открытые скобки иначе ошибка
        {
         ({ state, lexem } = VariantOperatorEnd(lexems, lexem, state,
charact[i]));
         skobch-=1;// вычитаем уже закрытую скобку
        }
        else ({ state, lexem } = VariantDefault(lexems, lexem, "error",
charact[i]));
      } break;

    }
  }
  lexems.add(state, lexem)
  lexems.print();
}

lexer(filePath);

//_____________________________________________________________________
// Функция предназначенная для поведения по умолчанию у большинства состояний
function VariantDefault(lexems, current_lexem, state, char) {
  switch (true) {
    case ':' == char: { // Завершение текущей лексемы и переход в состояние
поиска операторо присваивания
      lexems.add(state, current_lexem);
      return { state: ':', lexem: char };
    }
    case '(' == char: { // Завершение текущей лексемы и переход в состояние
поиска операторо присваивания
        lexems.add(state, current_lexem);
        return { state: '(', lexem: char };
      }
      case ')' == char: { // Завершение текущей лексемы и переход в состояние
поиска операторо присваивания
        lexems.add(state, current_lexem);
        return { state: ')', lexem: char };
      }
    case Alldels.includes(char): { // Завершение текущей лексемы и переход в
состояние поиска следующей лексемы
      lexems.add(state, current_lexem);
      return { state: 'del', lexem: char.trim() };
    }
    case Alphabet.includes(char): return { // Продолжение текущей лексемы и
переход в состояние поиска идентификатора
      state: 'identifier', lexem: current_lexem + char
    };
```

```
    default: { // Завершение текущей лексемы и сообщение о ошибке, переход в
состояние поиска новой лексемы
      lexems.add('error', current_lexem + char);
      return { state: 'del', lexem: '' };
    }
  }
}
// Функция предназначенная для поведения по умолчанию у состояний имеющих
один исключительный вариант
// Исключительный вариант задается двумя последними параметрами
function VariantExclude(lexems, current_lexem, state, char, exl_char,
exl_state) {
  switch (true) {
    case exl_char == char: { current_lexem += char; state = exl_state; };
break;
    default: ({ state, lexem:current_lexem } = VariantDefault(lexems,
current_lexem, state, char));
  }
  return { state, lexem: current_lexem };
}
// Функция используемая для чтения файла
async function getcharact(filePath) {
  return new Promise((resolve, reject) => {
    fs.readFile(filePath, 'utf8', (err, data) => {
      if (err) {
        console.error('Ошибка при чтении файла:', err);
        reject(err);
        return;
      }
      resolve(data.split(''));
    });
  });
}

// Функция предназначенная для поведения по умолчанию в местах где
гарантированно заканчивается лексема оператор
function VariantOperatorEnd(lexems, current_lexem, state, char) {
  lexems.add(state, current_lexem);
  current_lexem = char.trim();
  switch (true) {
    case '#' == char: state = '#'; break;
    case Alldels.includes(char): state = 'del'; break;
    case numbers.includes(char): state = 'number'; break;
    case Alphabet.includes(char): state = 'identifier'; break;
    default: {
      lexems.add('error', char);
      current_lexem = '';
      state = 'del';
    } break;
  }
  return { state, lexem: current_lexem };
}
```

7. Вывод

Были изучены основные понятия теории регулярных языков и грамматик, было проведено ознакомление с назначением и принципами работы конечных автоматов (КА) и лексических анализаторов (сканеров).

Были получены практические навыки построения КА на основе заданной регулярной грамматики. Были получены практические навыки построения сканера на примере заданного простейшего входного языка.