ГУАП

КАФЕДРА

№44

ОТЧЕТ
ЗАЩИЩЁН С ОЦЕНКОЙ
ПРЕПОДАВАТЕЛЬ

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

ОТЧЁТ О ЛАБОРАТОРНОЙ РАБОТЕ

ПОСТРОЕНИЕ СИНТАКСИЧЕСКОГО АНАЛИЗАТОРА И ПРОСТЕЙШЕГО ДЕРЕВАВЫВОДА

по курсу: СИСТЕМНОЕ ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ

РАБОТУ ВЫПОЛНИЛ

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

Санкт-Петербург 2024

Цель работы.

Изучение основных понятий теории грамматик простого и операторного предшествования, ознакомление с алгоритмами синтаксического анализа (разбора) для некоторых классов КС-грамматик, получение практических навыков создания простейшего синтаксического анализатора для заданной грамматики операторного предшествования. Получение практических навыков создания простейшего синтаксическогоанализатора для заданной грамматики операторного предшествования, обработка и представление результатов синтаксического анализа.

Задание по лабораторной работе.

Вариант 14.

Грамматика 2.

В список допустимых лексем входят: Идентификаторы, шестнадцатеричные числа.

Требуется написать программу, которая выполняет лексический анализ входного текста в соответствии с заданием, порождает таблицу лексем и выполняет синтаксический разбор текста по заданной грамматике. Текст на входном языке задается в виде символьного (текстового) файла. Допускается исходить из условия, что текст содержит не более одного предложения входного языка.

Запись заданной грамматики входного языка в формеБэкуса-Наура.

Язык G({S, F, T, E}, {:=, or, xor, and, not, (,), ;, a}, P, S):

$S \rightarrow \mathbf{a} := F;$
$F \rightarrow F\ \mathbf{or}\ T \mid F\ \mathbf{xor}\ T \mid T$
$T \rightarrow T\ \mathbf{and}\ E \mid E$
$E \rightarrow (F) \mid \mathbf{not}\ (F) \mid \mathbf{a}$

Множества крайних правых и крайних левых символовс указанием шагов построения.

Символ	L(0)	R(0)	L(1)	R(1)
E	(, not, a	), a	(, not, a	), a
T	T, E	E	T,E,(,not,a	E,), a
F	F,T	T	T, F, E, not, (, a	T, E,), a
S	a	;	a	;

Множества крайних правых и крайних левых терминальныхсимволов.

Символ	Lt(0)	Rt(0)	Lt(1)	Rt(1)
E	(, not, a	), a	(, not, a	), a
T	and	and	and,(,not,a	and,), a
F	or,xor	or,xor	or,xor,and, not, (, a	or,xor,and ,), a
S	a	;	a	;

Заполненная матрица предшествования для грамматики.

По правилам из методического пособия заполняем таблицу.

символы	a	:=	or	xor	and	not	(	)	;	$\perp k$
a	.>	=.	=.	=.	=.	=.	=.	.>	.>	
:=	<.					<.	<.		=.	
or	<.		=.	=.	=.	<.	<.			
xor	<.		=.	=.	=.	<.	<.			
and	<.		=.	=.	=.	<.	<.			
not							=.			
(	<.		=.	=.	=.	<.	<.	=.		
)	.>							.>		
;	.>						.>			.>
⊥n	<.									

Запись исходной грамматики с одним не терминалом

E → a := E;

E →E or E | E xor E | E

E →E and E | E

E → (E) | not(E) | a

Пример выполнения разбора простейшего предложения.

Входная цепочка: a:= (x21 xor 16D) or not(23A and x3);

Входная строка	Стек	Действие
a:= (x21 xor 16D) or not(23A and x3);⊥k	⊥n	÷п
:= (x21 xor 16D) or not(23A and x3);⊥k	⊥n a	÷с
(x21 xor 16D) or not(23A and x3);⊥k	⊥n E:=	÷п
x21 xor 16D) or not(23A and x3);⊥k	⊥n E:=(	÷п
xor 16D) or not(23A and x3);⊥k	⊥n E:=(x21	÷п
16D) or not(23A and x3);⊥k	⊥n E:=(x21 xor	÷п
) or not(23A and x3);⊥k	⊥n E:=(x21 xor 16D	÷с
or not(23A and x3);⊥k	⊥n E:=(E)	÷с
not(23A and x3);⊥k	⊥n E:=E or	÷п
(23A and x3);⊥k	⊥n E:=E or not	÷п
23A and x3);⊥k	⊥n E:=E or not(	÷п
and x3);⊥k	⊥n E:=E or not(23A	÷п
x3);⊥k	⊥n E:=E or not(23A and	÷п
);⊥k	⊥n E:=E or not(23A and x3	÷с
;⊥k	⊥n E:=E or not(E)	÷с
;⊥k	⊥n E:=E or E	÷с
⊥k	⊥n E:=E;	÷с
⊥k	⊥n E	

Пример построения дерева вывода.

На рисунке 1 представлено синтаксическое дерево для входной цепочки,разобранной выше

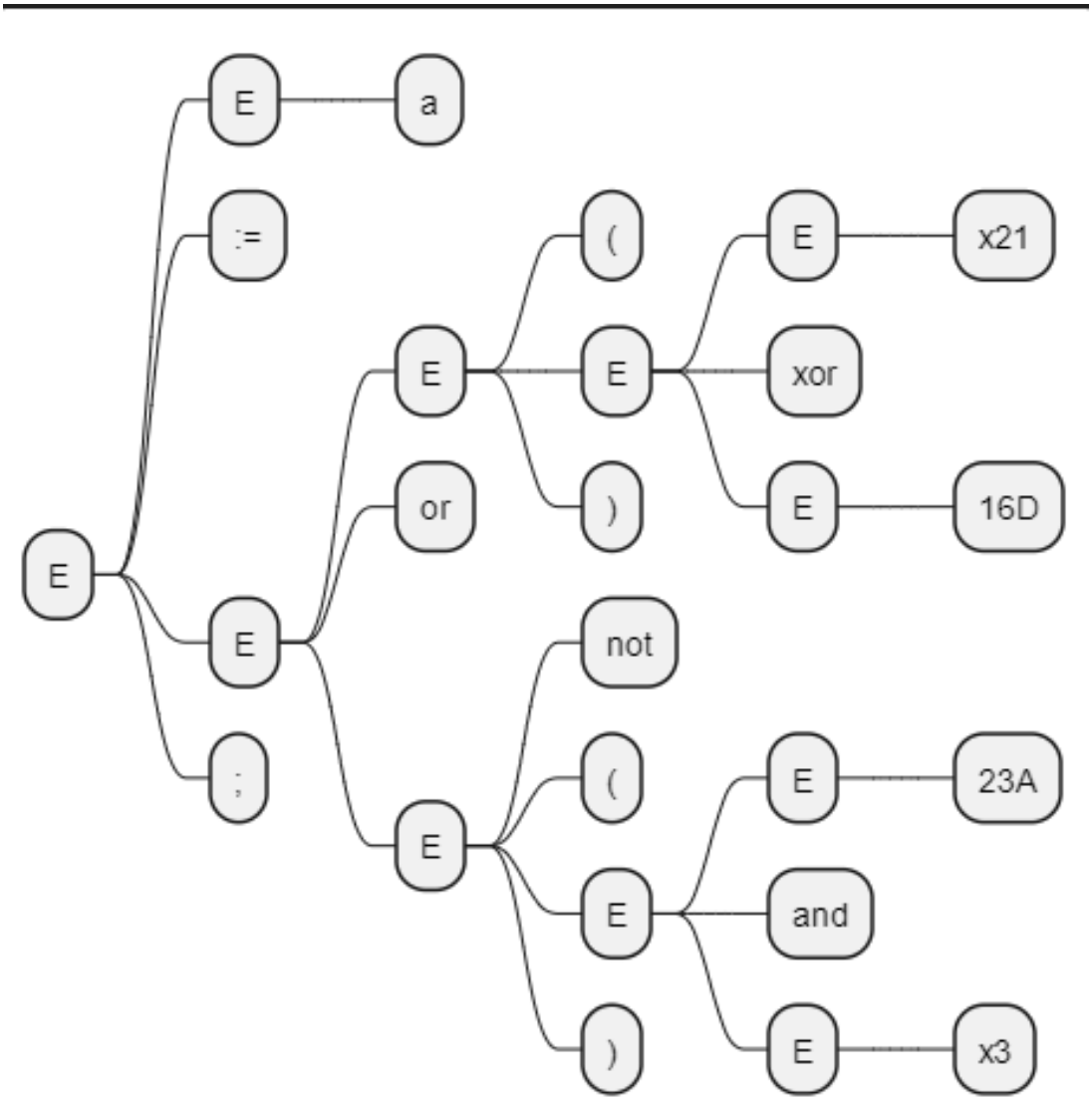

Рисунок 1 – Дерево вывода для входной цепочки

Текст программы

Подключаем первую лабораторную работу с лексемами, как модуль, ко второй.

Листинг кода:

Код лабораторной работы 1 (изменённый)

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

```
const fs = require('fs');

// 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);
  }
  getArray() {
    return this.lexems
  }
}
// Карта типов для лексем завершающихся в каком либо из состояний
const typeMap = {
  'del': 'del',
  'number': 'number',
  'r': 'or',
  'r1': 'xor',
  'd': 'and',
  't': 'not',
  ':=': 'assignment',
  '(': 'skobka',
  ')': 'skobka',
  'error': 'error',
  ':': 'error',
};



// Основная функция выступающая в качестве лексера
export default async function lexer(filePath) {
  // 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)
  return lexems;
}



//_____________________________________________________________
// 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 };
}
```

Код 2 лабораторной работы

```
const fs = require('fs');  // Получаем доступ к файловой системе
import lexer from "./lab1"; // Подключаем лексер, реализованный в прошлой лабораторной работе

const lexems = await lexer('./тест.txt'); // Генерируем список лексем из файла 'тест.txt'
const ArrayLexems1 = lexems.getArray();
const ArrayLexems = ArrayLexems1.map(it => {
  // Преобразуем лексемы: заменяем тип "number" на "identifier"
  if (it.type == "number") return { type: "identifier", lexem: it.lexem };
  return it;
});
// Проверяем, есть ли ошибки среди лексем, и выводим сообщение об ошибке
if (ArrayLexems.some(it => it.type === "error")) console.error("содержатся ошибки");
// разбираем все правила нашей грамматики
const rules = [
    { sver: "E", posl: ["identifier"] },//E -> a
    { sver: "E", posl: ["E", "and", "E"] },//E -> T and E
    { sver: "E", posl: ["E", "xor", "E"] },//E -> F xor T
    { sver: "E", posl: ["E", "assignment", "E","del"] },//E -> a := F;
    { sver: "E", posl: ["skobka", "E", "skobka"] },//E -> (F)
    { sver: "E", posl: ["not", "E"] },//E -> not (F)
    { sver: "E", posl: ["E", "or", "E"] },//E -> F or T
];

const Sver = []; // Объявляем массив сверток

let Tr = { // Создаем корень будущего дерева
  type: "NonTerminal",// обозначение нетерминалов
  ch: []//что входит внутрь
};

for (let i = 0; i < ArrayLexems.length; i++) {
```

```
  Sver.push(ArrayLexems[i]); // По одной добавляем лексемы в массив для свертки
  Tr.ch.push({ type: "Terminal", lexem: ArrayLexems[i].lexem }); // Добавляем новый лист в
дерево

  rules.forEach(it => { // Для каждого правила проверяем
    const element = it.posl; // Получаем последовательность, задающую правило
    if (element.length > Sver.length) return; // Если последовательность больше количества
элементов в свертке, то пропускаем правило

    // Сверяем последние элементы массива свертки с последовательностью правила
    for (let j = 0; j < element.length; j++) {
      if (Sver[Sver.length - 1 - j].type != element[element.length - 1 - j]) return; // Если
найдено несовпадение, переходим к следующей лексеме
    }

    let word = ""; // Создаем переменную для сохранения текста лексем
    // Записываем объединенную лексему
    for (let e = 0; e < element.length; e++) word = Sver.pop().lexem + " " + word.trim();

    // Создаем новую ветку дерева
    Tr = {
      type: "NonTerminal",
      ch: [
        ...Tr.ch.slice(0, Tr.ch.length - element.length), // Элементы до свертки оставляем на
том же уровне
        { type: "NonTerminal", lexem: word, ch: [...Tr.ch.slice(Tr.ch.length - element.length)]
} // Элементы свертки выносим в отдельную ветвь
      ]
    };

    // Записываем свернутые лексемы в виде свертки обратно в массив сверток
    Sver.push({ type: it.sver, lexem: word });
  });
}

// Проверяем, что выражение свернулось корректно
if (Sver.length === 1) console.log("Выражение корректно");

// Избавляемся от излишней вложенности
Tr= Tr.ch[0];

// Конвертируем получившийся объект в JSON для дальнейшей записи и хранения
const jsonString = JSON.stringify(Tr, null, 2);

// Записываем ответ в файл
await fs.writeFile('Tr.json', jsonString, (err) => {
  if (err) {
    console.error('Ошибка записи в файл', err);
  } else {
    console.log('Дерево сохранено');
  }
});
```

В консоли можно увидеть финальное состояние стека - единственный нетерминальный символ, содержащий внутри себя прочие входные символы и прочие нетерминалы.

Результат работы программы:


```
{
  "type": "NonTerminal",
  "lexem": "a  := ( x21  xor 16D ) or not ( 23A  and x3 ) ;",
  "ch": [
    {
      "type": "NonTerminal",
      "lexem": "a ",
      "ch": [
        {"type": "Terminal", "lexem": "a"}
      ]
    },
    {"type": "Terminal", "lexem": ":="},
    {
      "type": "NonTerminal",
      "lexem": "( x21  xor 16D ) or not ( 23A  and x3 )",
      "ch": [
        {
          "type": "NonTerminal",
          "lexem": "( x21  xor 16D )",
          "ch": [
            {"type": "Terminal", "lexem": "("},
            {
              "type": "NonTerminal",
              "lexem": "x21  xor 16D",
              "ch": [
                {"type": "NonTerminal", "lexem": "x21 ", "ch": [{"type": "Terminal",
"lexem": "x21"}]},
                {"type": "Terminal", "lexem": "xor"},
                {"type": "NonTerminal", "lexem": "16D ", "ch": [{"type": "Terminal",
"lexem": "16D"}]}
              ]
            },
            {"type": "Terminal", "lexem": ")"}
          ]
        },
        {"type": "Terminal", "lexem": "or"},
        {
          "type": "NonTerminal",
          "lexem": "not ( 23A  and x3 )",
          "ch": [
            {"type": "Terminal", "lexem": "not"},
            {
              "type": "NonTerminal",
              "lexem": "( 23A  and x3 )",
              "ch": [
                {"type": "Terminal", "lexem": "("},
                {
                  "type": "NonTerminal",
                  "lexem": "23A  and x3",
                  "ch": [
                    {"type": "NonTerminal", "lexem": "23A ", "ch": [{"type": "Terminal",
"lexem": "23A"}]},
                    {"type": "Terminal", "lexem": "and"},
```

```
                    {"type": "NonTerminal", "lexem": "x3 ", "ch": [{"type": "Terminal",
"lexem": "x3"}]}
                ]
            },
            {"type": "Terminal", "lexem": ")"}
        ]
    }
  ]
 }
 ]
 },
 {"type": "Terminal", "lexem": ";"}
]
}
```


Заключение.

Были изучены основные понятия теории грамматик простого и операторного предшествования, было проведено ознакомление с алгоритмами синтаксического анализа (разбора) для некоторых классов КС-грамматик, были получены практические навыки создания простейшего синтаксического анализатора для заданной грамматики операторного предшествования. Были получены практические навыки создания простейшего синтаксического анализатора для заданной грамматики операторного предшествования, были обработаны и представлены результаты синтаксического анализа.