САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО

Дисциплина: Методы трансляции

Отчет

по лабораторной работе №2

«Ручное построение нисходящих синтаксических анализаторов»

Выполнил(а): Яндаров Идрис Салманович

Номер ИСУ: 289646

студ. гр. М33351

Санкт-Петербург

2021

Задание. Вариант 5. Логические формулы в стиле Си

Логические формулы. Используются операции &, |, ^, !. Приоритет операций стандартный. Скобки могут использоваться для изменения приоритета. В качестве операндов выступают переменные с именем из одной буквы. Используйте один терминал для всех переменных. Для каждой логической операции должен быть заведен один терминал.

Разработка грамматики

Построим грамматику:

E -> O|E

E -> O

O -> X^O

O -> X

X -> A&X

X -> A

A -> var

A -> !A

A -> (E)

Нетерминал	**Описание**
E	Правильная логическая формула в стиле Си.
O	В случае наличия в *E* операции | ее явл-ся первым операндом. Иначе повторяет *E*.
X	В случае наличия в *O* операции ^ ее явл-ся первым операндом. Иначе повторяет *O*.
A	В случае наличия в *X* операции & ее явл-ся первым операндом. Иначе повторяет *X*. Может быть переменной,

	правильной логической формулой в скобках либо ее отрицанием.

В грамматике есть правое ветвление. Устраним ее. Получится грамматика:

E -> OE′

E′ -> |E

E′ -> ε

O -> XO′

O′ -> ^O

O′ -> ε

X -> AX′

X′ -> &X

X′ -> ε

A -> var

A -> !A

A -> (E)

Нетерминал	**Описание**
E	Правильная логическая формула в стиле Си.
E′	Продолжение *E* в случае наличия в нем операции |, иначе *ε*.
O	В случае наличия в *E* операции | явл-ся первым операндом. Иначе повторяет *E*.
O′	Продолжение *O* в случае наличия в нем операции ^, иначе *ε*.
X	В случае наличия в *O* операции ^ явл-ся первым операндом. Иначе повторяет *O*.
X′	Продолжение *X* в случае наличия в нем операции &,

	иначе ε.
A	В случае наличия в X операции & явл-ся первым операндом. Иначе повторяет X. Может быть переменной, правильной логической формулой в скобках либо ее отрицанием.

Построение лексического анализатора

Терминал	**Токен**
(	LBR
)	RBR
$	END
!	NOT
&	AND
^	XOR
|	OR
var	VAR

Код Token:

```
export enum Token {
 LBR = '(',
 RBR = ')',
 END = '$',
 NOT = '!',
 AND = '&',
 XOR = '^',
 OR = '|',
 VAR = 'var',
}
```

Код лексического анализатора:

```
import { Token } from './Token.js';

export default class LexicalAnalyzer {
 input: string;
 curChar: string;
```

```
curPos: number;
curToken: Token | undefined;

constructor(input: string) {
  this.input = input;
  this.curChar = '';
  this.curPos = 0;
  this.nextChar();
}

nextChar() {
  if (this.curPos === this.input.length) {
    this.curChar = '$';
    this.curPos++;
  } else if (this.curPos > this.input.length) {
    throw new Error("Can't execute nextChar(): end of input is reached");
  } else {
    this.curChar = this.input[this.curPos++];
  }
}

nextToken() {
  while (/\s/.test(this.curChar)) {
    this.nextChar();
  }

  if (/[a-zA-Z]/.test(this.curChar)) {
    this.nextChar();
    this.curToken = Token.VAR;
  } else {
    switch (this.curChar) {
      case '(':
        this.nextChar();
        this.curToken = Token.LBR;
        break;

      case ')':
        this.nextChar();
        this.curToken = Token.RBR;
        break;

      case '$':
        this.curToken = Token.END;
        break;

      case '!':
        this.nextChar();
        this.curToken = Token.NOT;
        break;

      case '&':
        this.nextChar();
        this.curToken = Token.AND;
        break;

      case '^':
        this.nextChar();
        this.curToken = Token.XOR;
        break;

      case '|':
        this.nextChar();
        this.curToken = Token.OR;
        break;

      default:
```

```
          throw new Error(
            `Can't execute nextToken(): got illegal character at ${this.curPos}`
          );
        }
      }
    }
  }
```

Построение синтаксического анализатора

Построим множества FIRST и FOLLOW для нетерминалов нашей грамматики.

Нетерминал	FIRST	FOLLOW
E	var, !, (	$,)
E'	|, ε	$,)
O	var, !, (	|, $,)
O'	^, ε	|, $,)
X	var, !, (	^, |, $,)
X'	&, ε	^, |, $,)
A	var, !, (	&, ^, |, $,)

Заведем структуру данных для хранения дерева.

```
import { digraph } from 'graphviz';
import { renderGraphFromSource } from 'graphviz-cli';
import { createDirectory, getAttrs } from './util.js';

export default class Tree {
 value: string;
 children: Array<Tree>;

 constructor(node: string, ...children: Array<Tree>) {
   this.value = node;
   this.children = children;
 }

 addChild(child: Tree) {
   this.children.push(child);
 }

 buildGraph() {
   const graph = digraph('G');
   const nodeToId = new Map<Tree, string>();
```

```
    let nodeId = 0;

    const addNode = (u: Tree) => {
      if (!nodeToId.has(u)) {
        nodeToId.set(u, String(nodeId++));
      }

      //@ts-ignore
      return graph.addNode(nodeToId.get(u), getAttrs(u));
    };

    const stack: Array<Tree> = [this];

    while (stack.length > 0) {
      //@ts-ignore
      const u: Tree = stack.pop();
      const uNode = addNode(u);

      u.children.forEach((v) => {
        const vNode = addNode(v);
        graph.addEdge(uNode, vNode);
        stack.push(v);
      });
    }

    return graph;
  }

  async visualizeTree(imageName: string) {
    createDirectory('images');

    await renderGraphFromSource(
      {
        input: this.buildGraph().to_dot(),
      },
      { format: 'png', name: `./images/${imageName}.png` }
    );
  }
}
```

Ниже приведен синтаксический анализатор с использованием рекурсивного спуска.

```
import LexicalAnalyzer from './LexicalAnalyzer.js';
import { Token } from './Token.js';
import Tree from './Tree.js';

export default class Parser {
  lex: LexicalAnalyzer | undefined;

  consume(c: string) {
    const consumedChar = this.lex?.input[this.lex?.curPos - 2];

    if (this.lex?.curToken !== c) {
      throw new Error(
        `Can't execute consume(): expected ${c}, but got ${this.lex?.curToken}${
          this.lex ? ` at ${this.lex?.curPos - 1}` : ``
        }`
      );
    }

    this.lex.nextToken();

    return consumedChar;
```

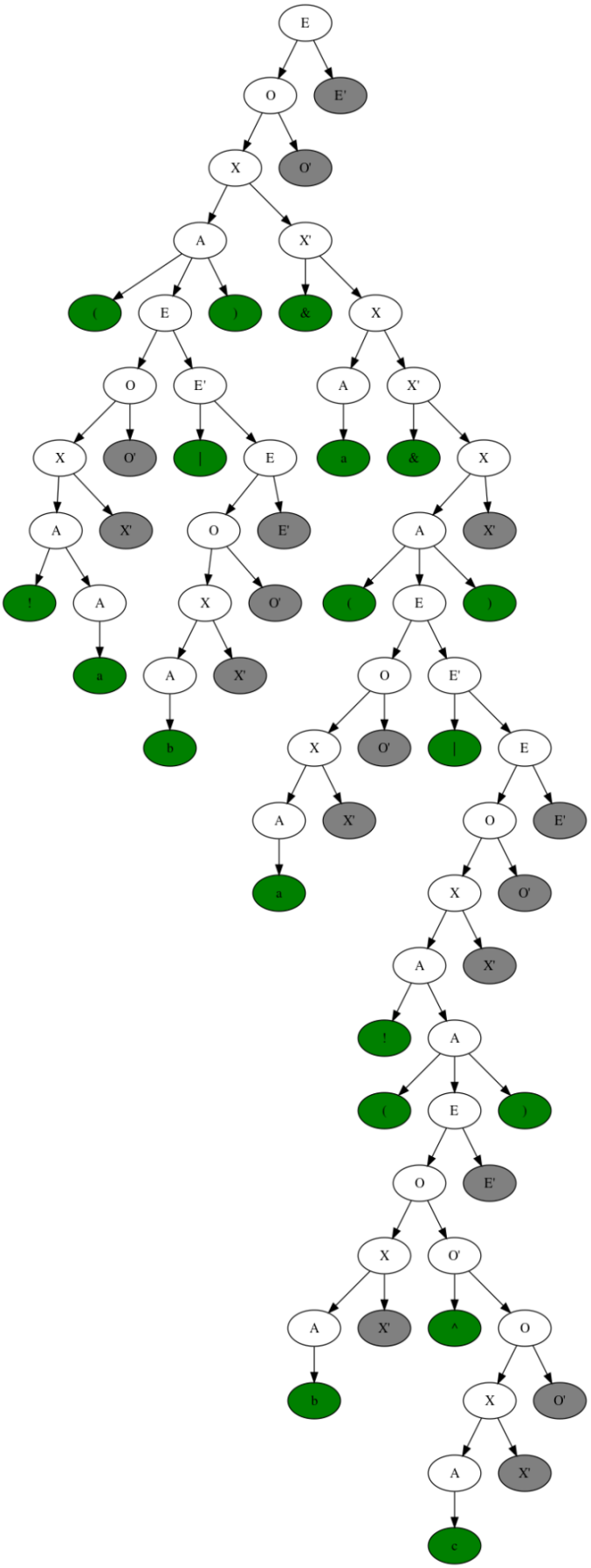
```
}

parse(input: string): Tree {
  this.lex = new LexicalAnalyzer(input);
  this.lex.nextToken();
  return this.E();
}

E(): Tree {
  const res = new Tree('E');

  switch (this.lex?.curToken) {
    case Token.VAR:
    case Token.NOT:
    case Token.LBR:
      res.addChild(this.O());
      res.addChild(this.E1());
      break;

    default:
      throw new Error(
        `Can't execute E(): got unexpected token ${this.lex?.curToken}${
          this.lex ? ` at ${this.lex?.curPos - 1}` : ``
        }`
      );
  }

  return res;
}

E1(): Tree {
  const res = new Tree("E'");

  switch (this.lex?.curToken) {
    case Token.OR:
      this.consume('|');
      res.addChild(new Tree('|'));
      res.addChild(this.E());
      break;

    case Token.END:
    case Token.RBR:
      break;

    default:
      throw new Error(
        `Can't execute E1(): got unexpected token ${this.lex?.curToken}${
          this.lex ? ` at ${this.lex?.curPos - 1}` : ``
        }`
      );
  }

  return res;
}

O(): Tree {
  const res = new Tree('O');

  switch (this.lex?.curToken) {
    case Token.VAR:
    case Token.NOT:
    case Token.LBR:
      res.addChild(this.X());
      res.addChild(this.O1());
      break;
```

```
    default:
      throw new Error(
        `Can't execute O(): got unexpected token ${this.lex?.curToken}${
          this.lex ? ` at ${this.lex?.curPos - 1}` : ``
        }`
      );
  }

  return res;
}

O1(): Tree {
  const res = new Tree("O'");

  switch (this.lex?.curToken) {
    case Token.XOR:
      this.consume('^');
      res.addChild(new Tree('^'));
      res.addChild(this.O());
      break;

    case Token.OR:
    case Token.END:
    case Token.RBR:
      break;

    default:
      throw new Error(
        `Can't execute O1(): got unexpected token ${this.lex?.curToken}${
          this.lex ? ` at ${this.lex?.curPos - 1}` : ``
        }`
      );
  }

  return res;
}

X(): Tree {
  const res = new Tree('X');

  switch (this.lex?.curToken) {
    case Token.VAR:
    case Token.NOT:
    case Token.LBR:
      res.addChild(this.A());
      res.addChild(this.X1());
      break;

    default:
      throw new Error(
        `Can't execute X(): got unexpected token ${this.lex?.curToken}${
          this.lex ? ` at ${this.lex?.curPos - 1}` : ``
        }`
      );
  }

  return res;
}

X1(): Tree {
  const res = new Tree("X'");

  switch (this.lex?.curToken) {
    case Token.AND:
      this.consume('&');
      res.addChild(new Tree('&'));
```

```
        res.addChild(this.X());
        break;

      case Token.XOR:
      case Token.OR:
      case Token.END:
      case Token.RBR:
        break;

      default:
        throw new Error(
          `Can't execute X1(): got unexpected token ${this.lex?.curToken}${
            this.lex ? ` at ${this.lex?.curPos - 1}` : ``
          }`
        );
    }

    return res;
  }

  A(): Tree {
    const res = new Tree('A');

    switch (this.lex?.curToken) {
      case Token.VAR:
        res.addChild(new Tree(this.consume('var') || 'var'));
        break;

      case Token.NOT:
        this.consume('!');
        res.addChild(new Tree('!'));
        res.addChild(this.A());
        break;

      case Token.LBR:
        this.consume('(');
        res.addChild(new Tree('('));
        res.addChild(this.E());
        this.consume(')');
        res.addChild(new Tree(')'));
        break;

      default:
        throw new Error(
          `Can't execute A(): got unexpected token ${this.lex?.curToken}${
            this.lex ? ` at ${this.lex?.curPos - 1}` : ``
          }`
        );
    }

    return res;
  }
}
```

Визуализация дерева разбора

Результатом являются *.png* картинки, находящиеся в папке *images* проекта и созданные с помощью *GraphViz*. Пример картинки:


E
O
E'
X
O'
A
X'
(
E
)
&
X
O
E'
A
X'
X
O'
|
E
a
&
X
A
X'
O
E'
A
X'
!
A
X
O'
(
E
)
a
A
X'
b
O
E'
X
O'
|
E
A
X'
O
E'
a
X
O'
A
X'
!
A
(
E
)
O
E'
X
O'
A
X'
^
O
b
X
O'
A
X'
c

Подготовка набора тестов

Тест	Описание
	Пустая строка. Грамматика такого не допускает. Не должно распарсить.
a & !b	Простое выражение. Должно распарсить.
!a & b ^ !c | d | (!(a | b) & !c)	Сложное выражение. Должен соблюдаться приоритет операций.
!a & b ^ !c | d || (!a | b)	Неверно составленное выражение. Не должно распарсить.
!a -> b	Наличие неверных символов. Не должно токенизировать.
!a ^ b	Наличие заглавных букв в качестве переменных. Должно распарсить.
(!a | b) & a & (a | !(b ^ c))	Пример выражения из условия. Должно распарсить.
!a ^b\n\n\n| c\t\t\t & !(b)	Много пробельных символов. Должно распарсить.