Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Московский государственный технический университет
имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

Факультет Информатика и системы управления
Кафедра «Программное обеспечение ЭВМ и информационные технологии»
ИУ-7

Лабораторная работа №4 «Синтаксический анализ операторного предшествования»

Выполнил студент: Агеев Алексей Владимирович

Группа: ИУ7-22М

Проверил: Андрей Алексеевич Ступников

Оценка: _____ Дата: _____ Подпись: _____

2020 год.

Цель. Изучение Восходящего синтаксического анализа на примере алгоритма приорита операторов.

Задачи

- разработать алгоритм синтаксического анализа приоритета операторов, который должен перводить входную строчку из стандартной записи в обратную польскую запись

Основная часть

Операторная грамматика

Грамматика G называется операторной грамматикой, если во множестве P отсутствуют эпсилон-правила, отсутствуют цеопчки, которые содержат последовательно 2 или более нетерминалов.

Отношение приоритетов

a < b - a “Уступает приоритет” b
a=b - a имеет тот же приоритет, что и b
a>b - a “забирает приоритет у”b

Задание (Варинат 2).

Исходная грамматика:

```
<expr> := <ar_expr><relation><ar_exp>
       | <ar_expr>
<ar_expr> := <ar_exp><sum_op><term>
          | <term>
<term> := <term><mul_op><factor>
       |<factor>
<factor> := <id>|<const>|(<ar_expr>)
<relation> := <|<=|==|<>|>|>=
<sum_op> := +|-
<mul_op> := *|/
<id> := $id
<const> = $const
```

После преобразований получаем следующую грамматику операторного предшествования:

```
<expr> := <expr>`<`<expr>
        | <expr>`<=`<expr>
        | <expr>`>`<expr>
        | <expr>`>=`<expr>
        | <expr>`==`<expr>
        | <expr>`<>`<expr>
        | <expr>`+`<expr>
        | <expr>`-`<expr>
        | <expr>`*`<expr>
        | <expr>`\`<expr>
        | `$id`
        | `$const`
        | `(`<expr>`)`
```

Исходя из свойств операций принятых в математике получаем следующую таблицу операторного предшествования:

	$id, $const	+,-	*,/	<,>,<>, <=,>=,==	(	)	$
$id, $const	0	>	>	>	1	>	>
+,-	<	>	<	>	<	>	>
*,/	<	>	>	>	<	>	>
<,>,<>, <=,>=,==	<	<	<	2	<	3	>
(	<	<	<	<	<	=	4
)	5	>	>	>	6	>	>
$	<	<	<	<	<	7	8

Обозначение ошибок

Номер ошибки	Описание
0	Между аргументами отсутсвует оператор
1	Меду аргументом и открывающейся скобкой должен быть оператор
2	Две или более операции отношения не могут идти подряд
3	После операции отношения не хватает арумента
4	Забыта закрывающая скобка
5	После закрывающей скобки длжен идти оператор
6	Между скобками необходимо оператор
7	Забыта открывающая скобка
8	Пустой ввод

Пример работы программы:
Вход: $id + $const <> $id*$const+$id
Выход: <> + $id * $const $id + $const $id

Вход: $id + $const <> $id*($const+$id)
Выход: <> * + $id $const $id + $const $id

Вход: $id + $const $id*($const+$id)
Выход: ERROR: 0 между аргументами отсутсвует оператор

Исходный код программы:

```
//
// Created by nrx on 23.05.2020.
//
```

```
#include "Lexer.h"
#include "Graph.h"
#include <fstream>
#include <iostream>
#include <vector>
#include <sstream>
#include <GraphToDOT.h>
#include <utility>
#include "Parser.h"

/*Таблица отношений приоритетов
*/std::vector<std::vector<char>> table =
        /*                                                              <, >,
          *                                                             <=, >=,
          *                           $id,$cons;   +, -;   *, /;   <>, ==;   (;   );     $*/
 /*$id, $const*/       {        {   '0',          '>',     '>',       '>',   '1','>', '>'},
/*+, -*/                          {   '<',          '>',     '<',       '>',   '<','>', '>'},
/**, /*/                         {   '<',          '>',     '>',       '>',   '<','>', '>'},
/*<, >, <=, >=, <>, ==*/   {   '<',          '<',     '<',       '2',   '<','3', '>'},
/*(*/                            {   '<',          '<',     '<',       '<',   '<','=', '4'},
/*)*/                            {   '5',          '>',     '>',       '>',   '6','>', '>'},
/*$*/                            {   '<',          '<',     '<',       '<',   '<','7', '8'},};



void errorHandler(char typeError) {
	std::cout << "ERROR: ";
	std::cout << typeError << " ";
	switch(typeError) {
		case '0':
			std::cout << "между аргументами отсутсвует оператор" <<
std::endl;
			break;
		case '1':
			std::cout   << "между аргументом и открывающейся скобкой
должен быть оператор" << std::endl;
			break;
		case '2':
			std::cout   << "две и более операций отношения не могут идти
подряд" << std::endl;
			break;
		case '3':
			std::cout   << "после операции отношения не хватает аргумента "
<< std::endl;
			break;

		case '4':
			std::cout << "Забыта заырывающая скобка" << std::endl;
			break;
		case '5':
```

```
                std::cout << "после закрывающей скобки должен идти оператор"
<< std::endl;
                break;
            case '6':
                std::cout << "между с      кобками необходим оператор" <<
std::endl;
                break;
            case '7':
                std::cout << "Забыта открывающая скобка" << std::endl;
            case '8':
                std::cout << "Пустой ввод" << std::endl;
        }

}

int main(int argc, char **argv){
        if(argc < 2) {
            std::cout << "Необходимо передать в качестве арумента путь к файлу"
<< std::endl;
            return -1;
        }
        std::ifstream file(argv[1]);
        if(!file.is_open()) {
            std::cout << "Не удалось открыть файл: " << argv[1] << std::endl;
            return -2;
        }
        std::string source;
        std::stringstream str;
        str << file.rdbuf();
        source = str.str();
        Lexer lexer(source);
        Parser parser(lexer, table, errorHandler);
//        if(!parser.run()) {
//            std::cout << "error" << std::endl;
//            return -3;
//        }
        parser.run();
        auto s = parser.get();
        std::cout << "Обратная польская запись: " << std::endl;
        while(!s.empty()) {
            Token t = s.top();
            s.pop();
            if(t.type != TokenType::ORBRACKET && t.type !=
TokenType::CRBRACKET) {
                std::cout << t.value << " ";
            }
        }
        std::cout << std::endl;
        return 0;
}
```

```
#include "Lexer.h"
#include "Token.h"
#include "Option.h"
#include <iostream>
#include <Lexer.h>

Lexer::Lexer(const std::string &text) {
	this->text = text;
	current.start = this->text.begin();
	current.end = current.start;
	current.start_column = 1;
	before = current;
	current_line = 1;
}

Lexer::Lexer(const Lexer &lexer) {
	text = lexer.text;
	current = lexer.current;
	before = lexer.before;
	current_line = lexer.current_line;
}

Token Lexer::makeToken(int type) {
	before = current;
	current.step();
	before_type = type;
	Token tok = {before_type, before.get()};
	before.end = before.start;
	return tok;
}

Option<LexerError> Lexer::isToken(const std::string &str) {
	bool isDetected = true;
	auto state = current.end;
	for(size_t i = 0; i < str.size(); i++) {
		if(current.end == text.end() || (++current.end) == text.end()) {
			current.end = state;
			return Option<LexerError>(END_OF_TEXT, true);
		}
		if(*current.end != str[i]) {
			isDetected = false;
			current.end = state;
			break;
		}
	}

	return Option<LexerError>(NOT_ERROR, isDetected);
```

```
}

bool Lexer::isSpace(const char &v) {
	if(v == '\n') {
		current_line += 1;
		current.start_column = 0;
		return	true;
	}
	return (v == ' ') || (v == '\t') || (v == '\r');
}

Token Lexer::get() {
	this->back();
	Token tok = this->next();
	return tok;
}

size_t Lexer::line() {
	return current_line;
}

size_t Lexer::column() {
	return before.start_column;
}

Token Lexer::next() {
	Token token;
	while((current.end != text.end())
		&& (isSpace(*current.end))) {
		current.step();
	}
	if ((current.end == text.end()) || (*current.end == '\0')) {
		return {TokenType::END, ""};
	}

	switch (*current.end) {
		case '(':
			token =	makeToken(TokenType::ORBRACKET);
			break;
		case ')':
			token = makeToken(TokenType::CRBRACKET);
			break;
		case '<':
			if(!(*(++current.end) == '=' || *current.end == '>'))
				--current.end;
			token =	makeToken(TokenType::RELATION);
			break;
		case '>':
			if(*(++current.end) != '=')
				--current.end;
```

```
                token = makeToken(TokenType::RELATION);
                break;
            case '=':
                if(*(++current.end) == '=') {
                    token = makeToken(TokenType::RELATION);
                    break;
                }
                token = makeToken(TokenType::UNDEFINED);
                break;
            case '+':
            case '-':
                token = makeToken(TokenType::SUM_OP);
                break;
            case '*':
            case '/':
                token = makeToken(TokenType::MUL_OP);
                break;
            case '$': {
                const std::string id("id");
                const std::string _const("const");
                Option<LexerError> res = isToken(id);
                if (res)    {
                    token =    makeToken(TokenType::ID);
                    break;
                }
                if(res.get() != NOT_ERROR) {
                    token = makeToken(TokenType::UNDEFINED);
                }
                res = isToken(_const);
                if(res) {
                    token =    makeToken(TokenType::CONST);
                    break;
                }
                token = makeToken(TokenType::UNDEFINED);
                break;
            }

            default:
                token =    makeToken(TokenType::UNDEFINED);
        }

        return token;

}

void Lexer::back() {
        current = before;
}

void Lexer::token_itr::step() {
```

```
        end++;
        start_column += std::distance(start, end);
        start = end;
}

std::string Lexer::token_itr::get() {
        std::string res;
        std::string::iterator tmp = end;
        tmp++;
        std::copy(start, tmp, std::back_inserter(res));
        return res;
}

//
// Created by nrx on 23.05.2020.
//

#include "Lexer.h"
#include <vector>
#include "Parser.h"



Parser::Parser(Lexer l, std::vector<std::vector<char>> t, void (*errorHandler)(char)) :
lexer(l), table(t), errorHandler(errorHandler)
{}

bool Parser::run() {
        const Token NT = {TokenType::NonTerminal, "E"};
        Token b = lexer.next();
        std::stack<Token> stack;
        stack.push({TokenType::END, "$"});
        Token a = stack.top();
        while(true) {
                if((stack.top().type == TokenType::END) && (b.type ==
TokenType::END)) {
                        return true;
                }
                if((table[a.type][b.type] == '<') || (table[a.type][b.type] == '=')) {
                        stack.push(b);
                        b = lexer.next();
                        a = stack.top();
                } else {
                        if(table[a.type][b.type] == '>') {
                                Token c;
                                std::cout << "Свертка: ";
                                do {
                                        c = stack.top();
                                        stack.pop();
```

```
                    a = stack.top();
                    conv.push(c);
                    std::cout <<    c.value <<    " " ;
                }while( (c.type == TokenType::NonTerminal)||
                ((stack.size() > 1) && (table[a.type][c.type] != '<')));
                std::cout << std::endl;
                //conv.push(node);
            } else {
                errorHandler(table[a.type][b.type]);
                return false;
            }
        }
        auto stack_print = stack;
        while(!stack_print.empty()) {
            auto el = stack_print.top();
            stack_print.pop();
            std::cout << el.value << "; ";
        }
        std::cout << std::endl;


    }
    return true;
}

std::stack<Token> Parser::get() {
    return conv;
}
```