Національний Технічний Університет України

“Київский Політехнічний Інститут”

Факультет Інформатики та Обчислювальної Техніки

Кафедра обчислювальної техніки

**Лабораторна робота №4**

по курсу «Системне програмування»

Виконав:

студент 3 курсу

ФІОТ гр. ІО-92

Петрук Вадим

Киев 2011

ПОБУДОВА СИНТАКСИЧНОГО АНАЛІЗАТОРА

**Лістинг програми:**

**package com.lab536.lab4;**

**import java.util.ArrayList;**

**import java.util.Arrays;**

**import com.lab536.lab1.\*;**

**import com.lab536.lab2.AbstractPrimitive;**

**import com.lab536.lab2.NonTerminalPrimitive;**

**import com.lab536.lab2.Operations;**

**import com.lab536.lab2.TerminalPrimitive;**

**import com.lab536.lab3.Token;**

**import com.lab536.lab3.enums.OperatorSymbols;**

**import com.lab536.lab3.enums.SeparatorSymbols;**

**public class Parser {**

**private NonTerminalPrimitive<?> tree;**

**private DataTable input;**

**private ArrayList<AbstractPrimitive<?>> subject;**

**private TableWrapper tables = TableWrapper.getInstance();**

**private int top = 0;**

**private int tokenTop = 0;**

**private int pointer = 0;**

**private int[] stack = new int[255];**

**private Object[] tokenStack = new Object[255];**

**public NonTerminalPrimitive<?> getTree(){**

**return tree;**

**}**

**public Parser(DataTable dt){**

**if((dt != null) && !(dt.getAll().length == 0)){**

**input = dt;**

**}**

**else{**

**throw new IllegalArgumentException("Can't accept empty table");**

**}**

**initSubject();**

**}**

**private void initSubject(){**

**subject = new ArrayList<AbstractPrimitive<?>>();**

**for(byte i = 0; i < input.getAll().length; i++){**

**Object o = input.get(i).getValue("operator");**

**if(o.getClass().isEnum()){**

**if(!((o.getClass() == SeparatorSymbols.class)&&((SeparatorSymbols)o == SeparatorSymbols.SPACE))){**

**TerminalPrimitive<?> operWrapper = new TerminalPrimitive<Operations>((Operations)o);**

**subject.add(operWrapper);**

**}**

**}**

**else{**

**RuleHandleDecorator<Token> tokenWrapper = new RuleHandleDecorator<Token>((Token) o);**

**subject.add(tokenWrapper);**

**}**

**}**

**subject.add(new TerminalPrimitive<String>("$"));**

**}**

**public void parse(){**

**stack[0] = 0;**

**Object a = subject.get(0).getContent();**

**tokenStack[0] = subject.get(0);**

**boolean proceed = true;**

**while (proceed){**

**int s = stack[top];**

**String action = tables.action(s, a);**

**if(!tables.isDilemma(action)){**

**if(tables.isShift(action)){**

**a = shift(tables.getState(action));**

**} else if (tables.isReduce(action)){**

**reduce(tables.getState(action));**

**} else if (tables.isAccept(action)){**

**proceed = false;**

**} else {**

**System.out.println("Error on position "+pointer+" at "+subject.get(pointer)+". Proceeding to the closest separator(\";\")");**

**boolean flag = false;**

**while((pointer < subject.size() - 1)&&(!flag)){**

**pointer++;**

**if(subject.get(pointer).getContent() == SeparatorSymbols.SEMICOLON){**

**flag = true;**

**}**

**}**

**if(!flag){**

**return;**

**}**

**pointer++;**

**top = 0;**

**tokenTop = 0;**

**}**

**}**

**else{**

**if(action.equals("r3/s6")){ // Check next AO priority.**

**if(tokenStack[tokenTop - 1].getClass() == OperatorSymbols.class){**

**boolean f = ((OperatorSymbols) a).priority >**

**((OperatorSymbols) tokenStack[tokenTop - 1]).priority;**

**if(f){**

**reduce(3);**

**} else{**

**shift(6);**

**}**

**}**

**} else if (action.equals("r9/r10")){ // Check operation.**

**if(((AbstractPrimitive<?>)tokenStack[tokenTop - 2]).getContent() == RuleTypes.EXPRESSION){**

**reduce(9);**

**} else if(((AbstractPrimitive<?>)tokenStack[tokenTop - 2]).getContent() == RuleTypes.L\_EXPRESSION){**

**reduce(10);**

**}**

**} else{**

**throw new IllegalArgumentException("0\_o");**

**}**

**}**

**}**

**tree = new NonTerminalPrimitive<Object>(tokenStack[0]);**

**}**

**private void reduce(int state) {**

**// pointer--;**

**Object[] rule = tables.getRule(state);**

**int reduceLength = rule.length - 1;**

**top -= reduceLength;**

**tokenTop -= reduceLength;**

**stack[top + 1] = tables.gotoValue(stack[top], (RuleTypes)rule[0]);**

**top++;**

**boolean corresponds = true;**

**for(int i = 0; i < reduceLength; i++){**

**if(tokenStack[i + tokenTop].getClass().isEnum()){**

**if(tokenStack[i + tokenTop] != rule[i+1]){**

**corresponds = false;**

**}**

**} else if(((AbstractPrimitive<?>)tokenStack[i + tokenTop]).getContent() !=**

**rule[i+1]){**

**corresponds = false;**

**}**

**}**

**if(!corresponds){**

**System.out.println("Syntax error on position "+pointer+" at "+subject.get(pointer)+". Proceeding to the closest separator(\";\")");**

**} else{**

**NonTerminalPrimitive<?> composite = tables.compose(rule,**

**Arrays.copyOfRange(tokenStack,**

**tokenTop,**

**tokenTop + reduceLength));**

**tokenStack[tokenTop] = new RuleHandleDecorator<NonTerminalPrimitive<?>>(**

**composite, (RuleTypes)rule[0]);**

**tokenTop++;**

**tokenStack[tokenTop] = tokenStack[tokenTop + reduceLength - 1];**

**}**

**}**

**private Object shift(int state) {**

**top++;**

**stack[top] = state;**

**pointer++;**

**Object a = subject.get(pointer).getContent();**

**tokenTop++;**

**tokenStack[tokenTop] = subject.get(pointer);**

**return a;**

**}**

**}**

package com.lab536.lab4;

import java.util.ArrayList;

import java.util.HashMap;

import java.util.Map;

import com.lab536.lab2.NonTerminalPrimitive;

import com.lab536.lab2.TerminalPrimitive;

import com.lab536.lab3.enums.KeyWord;

import com.lab536.lab3.enums.OperatorSymbols;

import com.lab536.lab3.enums.SeparatorSymbols;

public class TableWrapper {

private ArrayList<HashMap<Object, String>> actionsTable = new ArrayList<HashMap<Object, String>>();

private ArrayList<HashMap<RuleTypes, Integer>> gotoTable = new ArrayList<HashMap<RuleTypes, Integer>>();

private ArrayList<Object[]> rules = new ArrayList<Object[]>();

private static TableWrapper singleton;

@SuppressWarnings("serial")

private TableWrapper() {

actionsTable.add(new HashMap<Object, String>() {

{

put(RuleTypes.VARIABLE, "s23");

put(KeyWord.IF, "s5");

put(KeyWord.REPEAT, "s16");

}

});

gotoTable.add(new HashMap<RuleTypes, Integer>() {

{

put(RuleTypes.OPERATION, 2);

put(RuleTypes.OPER\_LIST, 1);

}

});

actionsTable.add(new HashMap<Object, String>() {

{

put(SeparatorSymbols.SEMICOLON, "s3");

put("$", "ACC");

}

});

gotoTable.add(new HashMap<RuleTypes, Integer>() {

{

put(RuleTypes.OPERATION, 2);

}

});

actionsTable.add(new HashMap<Object, String>() {

{

put(RuleTypes.VARIABLE, "r11");

put(KeyWord.UNTIL, "r11");

put(KeyWord.IF, "r11");

put(KeyWord.REPEAT, "r11");

put("$", "r11");

}

});

gotoTable.add(new HashMap<RuleTypes, Integer>(){

{

put(RuleTypes.OPERATION, 2);

put(RuleTypes.OPER\_LIST, 1);

}

});

actionsTable.add(new HashMap<Object, String>() {

{

put(RuleTypes.VARIABLE, "s23");

put(KeyWord.IF, "s5");

put(KeyWord.REPEAT, "s16");

}

});

gotoTable.add(new HashMap<RuleTypes, Integer>() {

{

put(RuleTypes.OPERATION, 4);

}

});

actionsTable.add(new HashMap<Object, String>() {

{

put("$", "r12");

}

});

gotoTable.add(new HashMap<RuleTypes, Integer>());

actionsTable.add(new HashMap<Object, String>() {

{

put(RuleTypes.VARIABLE, "s8");

put(RuleTypes.CONSTANT, "s9");

put(SeparatorSymbols.BRACKET\_OPENER, "s10");

}

});

gotoTable.add(new HashMap<RuleTypes, Integer>() {

{

put(RuleTypes.VARIABLE,8);

put(RuleTypes.EXPRESSION, 6);

put(RuleTypes.L\_EXPRESSION, 28);

}

});

actionsTable.add(new HashMap<Object, String>() {

{

put(SeparatorSymbols.BRACKET\_CLOSER, "s13");

put(OperatorSymbols.PLUS, "s11");

put(OperatorSymbols.MINUS, "s11");

put(OperatorSymbols.MUL, "s11");

put(OperatorSymbols.EQUALS, "s7");

put(OperatorSymbols.G\_EQUALS, "s7");

put(OperatorSymbols.L\_EQUALS, "s7");

put(OperatorSymbols.GREATER, "s7");

put(OperatorSymbols.LESSER, "s7");

}

});

gotoTable.add(new HashMap<RuleTypes, Integer>());

actionsTable.add(new HashMap<Object, String>() {

{

put(RuleTypes.VARIABLE, "s8");

put(RuleTypes.CONSTANT, "s9");

put(SeparatorSymbols.BRACKET\_OPENER, "s10");

}

});

gotoTable.add(new HashMap<RuleTypes, Integer>() {

{

put(RuleTypes.VARIABLE,8);

put(RuleTypes.EXPRESSION, 29);

}

});

actionsTable.add(new HashMap<Object, String>() {

{

put(SeparatorSymbols.SQUARE\_BRACKET\_OPENER, "s12");

put(SeparatorSymbols.SQUARE\_BRACKET\_CLOSER, "r1");

put(SeparatorSymbols.BRACKET\_CLOSER, "r1");

put(OperatorSymbols.PLUS, "r1");

put(OperatorSymbols.MINUS, "r1");

put(OperatorSymbols.MUL, "r1");

put(OperatorSymbols.EQUALS, "r1");

put(OperatorSymbols.G\_EQUALS, "r1");

put(OperatorSymbols.L\_EQUALS, "r1");

put(OperatorSymbols.GREATER, "r1");

put(OperatorSymbols.LESSER, "r1");

put(SeparatorSymbols.SEMICOLON, "r1");

put(KeyWord.THEN, "r1");

put("$", "r1");

}

});

gotoTable.add(new HashMap<RuleTypes, Integer>());

actionsTable.add(new HashMap<Object, String>() {

{

put(SeparatorSymbols.SQUARE\_BRACKET\_CLOSER, "r2");

put(SeparatorSymbols.BRACKET\_CLOSER, "r2");

put(OperatorSymbols.PLUS, "r2");

put(OperatorSymbols.MINUS, "r2");

put(OperatorSymbols.MUL, "r2");

put(OperatorSymbols.EQUALS, "r2");

put(OperatorSymbols.G\_EQUALS, "r2");

put(OperatorSymbols.L\_EQUALS, "r2");

put(OperatorSymbols.GREATER, "r2");

put(OperatorSymbols.LESSER, "r2");

put(SeparatorSymbols.SEMICOLON, "r2");

put(KeyWord.THEN, "r2");

put("$", "r2");

}

});

gotoTable.add(new HashMap<RuleTypes, Integer>());

actionsTable.add(new HashMap<Object, String>() {

{

put(RuleTypes.VARIABLE, "s8");

put(RuleTypes.CONSTANT, "s9");

put(SeparatorSymbols.BRACKET\_OPENER, "s10");

}

});

gotoTable.add(new HashMap<RuleTypes, Integer>() {

{

put(RuleTypes.VARIABLE,8);

put(RuleTypes.EXPRESSION, 6);

}

});

actionsTable.add(new HashMap<Object, String>() {

{

put(RuleTypes.VARIABLE, "s8");

put(RuleTypes.CONSTANT, "s9");

put(SeparatorSymbols.BRACKET\_OPENER, "s10");

}

});

gotoTable.add(new HashMap<RuleTypes, Integer>() {

{

put(RuleTypes.VARIABLE,8);

put(RuleTypes.EXPRESSION, 14);

}

});

actionsTable.add(new HashMap<Object, String>() {

{

put(RuleTypes.VARIABLE, "s8");

put(RuleTypes.CONSTANT, "s9");

put(SeparatorSymbols.BRACKET\_OPENER, "s10");

}

});

gotoTable.add(new HashMap<RuleTypes, Integer>() {

{

put(RuleTypes.VARIABLE,8);

put(RuleTypes.EXPRESSION, 21);

}

});

actionsTable.add(new HashMap<Object, String>() {

{

put(SeparatorSymbols.BRACKET\_CLOSER, "r4");

put(SeparatorSymbols.SQUARE\_BRACKET\_CLOSER, "r4");

put(OperatorSymbols.PLUS, "r4");

put(OperatorSymbols.MINUS, "r4");

put(OperatorSymbols.MUL, "r4");

put(OperatorSymbols.EQUALS, "r4");

put(OperatorSymbols.G\_EQUALS, "r4");

put(OperatorSymbols.L\_EQUALS, "r4");

put(OperatorSymbols.GREATER, "r4");

put(OperatorSymbols.LESSER, "r4");

put(SeparatorSymbols.SEMICOLON, "r4");

put(KeyWord.THEN, "r4");

put("$", "r4");

}

});

gotoTable.add(new HashMap<RuleTypes, Integer>());

actionsTable.add(new HashMap<Object, String>() {

{

put(SeparatorSymbols.BRACKET\_CLOSER, "r3");

put(SeparatorSymbols.SQUARE\_BRACKET\_CLOSER, "r3");

put(OperatorSymbols.PLUS, "r3/s6");

put(OperatorSymbols.MINUS, "r3/s6");

put(OperatorSymbols.MUL, "r3/s6");

put(OperatorSymbols.EQUALS, "r3/s6");

put(OperatorSymbols.G\_EQUALS, "r3/s6");

put(OperatorSymbols.L\_EQUALS, "r3/s6");

put(OperatorSymbols.GREATER, "r3/s6");

put(OperatorSymbols.LESSER, "r3/s6");

put(SeparatorSymbols.SEMICOLON, "r3");

put(KeyWord.THEN, "r3");

put("$", "r3");

}

});

gotoTable.add(new HashMap<RuleTypes, Integer>());

actionsTable.add(new HashMap<Object, String>());

gotoTable.add(new HashMap<RuleTypes, Integer>() {

{

put(RuleTypes.OPERATION, 2);

put(RuleTypes.OPER\_LIST, 30);

}

});

actionsTable.add(new HashMap<Object, String>(){

{

put(RuleTypes.VARIABLE, "s23");

put(KeyWord.IF, "s5");

put(KeyWord.REPEAT, "s16");

}

});

gotoTable.add(new HashMap<RuleTypes, Integer>() {

{

put(RuleTypes.OPERATION, 2);

put(RuleTypes.OPER\_LIST, 17);

}

});

actionsTable.add(new HashMap<Object, String>() {

{

put(KeyWord.UNTIL, "s18");

}

});

gotoTable.add(new HashMap<RuleTypes, Integer>());

actionsTable.add(new HashMap<Object, String>() {

{

put(RuleTypes.VARIABLE, "s8");

put(RuleTypes.CONSTANT, "s9");

put(SeparatorSymbols.BRACKET\_OPENER, "s10");

}

});

gotoTable.add(new HashMap<RuleTypes, Integer>(){

{

put(RuleTypes.VARIABLE,8);

put(RuleTypes.EXPRESSION,6);

put(RuleTypes.L\_EXPRESSION,19);

}

});

actionsTable.add(new HashMap<Object, String>() {

{

put(SeparatorSymbols.SEMICOLON, "s20");

}

});

gotoTable.add(new HashMap<RuleTypes, Integer>());

actionsTable.add(new HashMap<Object, String>() {

{

put(KeyWord.IF, "r8");

put(KeyWord.REPEAT, "r8");

put("$", "r8");

}

});

gotoTable.add(new HashMap<RuleTypes, Integer>());

actionsTable.add(new HashMap<Object, String>() {

{

put(SeparatorSymbols.SQUARE\_BRACKET\_CLOSER, "s22");

}

});

gotoTable.add(new HashMap<RuleTypes, Integer>());

actionsTable.add(new HashMap<Object, String>() {

{

put(SeparatorSymbols.BRACKET\_CLOSER, "r6");

put(OperatorSymbols.PLUS, "r6");

put(OperatorSymbols.MINUS, "r6");

put(OperatorSymbols.MUL, "r6");

put(OperatorSymbols.EQUALS, "r6");

put(OperatorSymbols.G\_EQUALS, "r6");

put(OperatorSymbols.L\_EQUALS, "r6");

put(OperatorSymbols.GREATER, "r6");

put(OperatorSymbols.LESSER, "r6");

put(SeparatorSymbols.SEMICOLON, "r6");

put(OperatorSymbols.ASSIGN, "r6");

put(KeyWord.THEN, "r6");

put("$", "r6");

}

});

gotoTable.add(new HashMap<RuleTypes, Integer>());

actionsTable.add(new HashMap<Object, String>() {

{

put(OperatorSymbols.ASSIGN, "s24");

}

});

gotoTable.add(new HashMap<RuleTypes, Integer>());

actionsTable.add(new HashMap<Object, String>() {

{

put(RuleTypes.VARIABLE, "s8");

put(RuleTypes.CONSTANT, "s9");

put(SeparatorSymbols.BRACKET\_OPENER, "s10");

}

});

gotoTable.add(new HashMap<RuleTypes, Integer>() {

{

put(RuleTypes.VARIABLE,8);

put(RuleTypes.EXPRESSION, 25);

}

});

actionsTable.add(new HashMap<Object, String>() {

{

put(OperatorSymbols.PLUS, "s11");

put(OperatorSymbols.MINUS, "s11");

put(OperatorSymbols.MUL, "s11");

put(OperatorSymbols.EQUALS, "s11");

put(OperatorSymbols.G\_EQUALS, "s11");

put(OperatorSymbols.L\_EQUALS, "s11");

put(OperatorSymbols.GREATER, "s11");

put(OperatorSymbols.LESSER, "s11");

put(SeparatorSymbols.SEMICOLON, "s27");

}

});

gotoTable.add(new HashMap<RuleTypes, Integer>() {

{

put(RuleTypes.EXPRESSION, 6);

}

});

actionsTable.add(new HashMap<Object, String>());

gotoTable.add(new HashMap<RuleTypes, Integer>());

actionsTable.add(new HashMap<Object, String>() {

{

put(KeyWord.REPEAT, "r9/r10");

put(KeyWord.UNTIL, "r9/r10");

put(KeyWord.IF, "r9/r10");

put("$", "r9/r10");

}

});

gotoTable.add(new HashMap<RuleTypes, Integer>(){

{

put(RuleTypes.OPERATION, 2);

put(RuleTypes.OPER\_LIST, 1);

}

});

actionsTable.add(new HashMap<Object, String>() {

{

put(KeyWord.THEN, "s15");

}

});

gotoTable.add(new HashMap<RuleTypes, Integer>());

actionsTable.add(new HashMap<Object, String>() {

{

put(SeparatorSymbols.SEMICOLON, "r5");

put(KeyWord.IF, "r5");

put("$", "r5");

}

});

gotoTable.add(new HashMap<RuleTypes, Integer>());

actionsTable.add(new HashMap<Object, String>() {

{

put(KeyWord.REPEAT, "r7");

put(KeyWord.IF, "r7");

put("$", "r7");

}

});

gotoTable.add(new HashMap<RuleTypes, Integer>(){

{

put(RuleTypes.OPERATION, 2);

put(RuleTypes.OPER\_LIST, 1);

}

});

rules.add(new Object[]{RuleTypes.EXPRESSION,RuleTypes.VARIABLE});

rules.add(new Object[]{RuleTypes.EXPRESSION,RuleTypes.CONSTANT});

rules.add(new Object[]{RuleTypes.EXPRESSION,

RuleTypes.EXPRESSION,

OperatorSymbols.PLUS,

RuleTypes.EXPRESSION

});

rules.add(new Object[]{RuleTypes.EXPRESSION,

SeparatorSymbols.BRACKET\_OPENER,

RuleTypes.EXPRESSION,

SeparatorSymbols.BRACKET\_CLOSER

});

rules.add(new Object[]{RuleTypes.L\_EXPRESSION,

RuleTypes.EXPRESSION,

OperatorSymbols.EQUALS,

RuleTypes.EXPRESSION

});

rules.add(new Object[]{RuleTypes.VARIABLE,

RuleTypes.VARIABLE,

SeparatorSymbols.SQUARE\_BRACKET\_OPENER,

RuleTypes.EXPRESSION,

SeparatorSymbols.SQUARE\_BRACKET\_CLOSER

});

rules.add(new Object[]{RuleTypes.OPERATION,

KeyWord.IF,

RuleTypes.L\_EXPRESSION,

KeyWord.THEN,

RuleTypes.OPER\_LIST,

SeparatorSymbols.SEMICOLON

});

rules.add(new Object[]{RuleTypes.OPERATION,

KeyWord.REPEAT,

RuleTypes.OPER\_LIST,

KeyWord.UNTIL,

RuleTypes.L\_EXPRESSION,

SeparatorSymbols.SEMICOLON

});

rules.add(new Object[]{RuleTypes.OPERATION,

RuleTypes.VARIABLE,

OperatorSymbols.ASSIGN,

RuleTypes.EXPRESSION,

SeparatorSymbols.SEMICOLON

});

rules.add(new Object[]{RuleTypes.OPERATION,

RuleTypes.VARIABLE,

OperatorSymbols.ASSIGN,

RuleTypes.L\_EXPRESSION,

SeparatorSymbols.SEMICOLON

});

rules.add(new Object[]{RuleTypes.OPER\_LIST,

RuleTypes.OPERATION

});

rules.add(new Object[]{RuleTypes.OPER\_LIST,

RuleTypes.OPER\_LIST,

SeparatorSymbols.SEMICOLON,

RuleTypes.OPERATION

});

}

public static TableWrapper getInstance() {

if (singleton == null) {

singleton = new TableWrapper();

}

return singleton;

}

public String action(int i, Object nextToken) {

Map<Object, String> m = actionsTable.get(i);

if(m.containsKey(nextToken)){

return m.get(nextToken);

}

return "e";

}

public int gotoValue(int t, RuleTypes type) {

Map<RuleTypes, Integer> m = gotoTable.get(t);

if(m.containsKey(type)){

return m.get(type);

}

return -1;

}

public Object[] getRule(int i){

return rules.get(i - 1);

}

public boolean isShift(String s){

return s.charAt(0) == 's';

}

public boolean isReduce(String s){

return s.charAt(0) == 'r';

}

public boolean isAccept(String s){

return s.equals("ACC");

}

public boolean isDilemma(String s){

return s.contains("/");

}

public int getState(String s){

return Integer.parseInt(s.substring(1));

}

public NonTerminalPrimitive<?> compose(Object[] rule, final Object[] toCompose) {

int index = rules.indexOf(rule);

switch(index){

case 0:

case 1:

return new NonTerminalPrimitive<Object>(toCompose[0]);

case 2:

case 3:

case 4:

case 11:

return new NonTerminalPrimitive<Object>(toCompose[1]) {

{

addLeftChild(new TerminalPrimitive<Object>(toCompose[0]));

addRightChild(new TerminalPrimitive<Object>(toCompose[2]));

}

};

case 5:

case 8:

case 9:

return new NonTerminalPrimitive<Object>(toCompose[0]) {

{

addRightChild(new NonTerminalPrimitive<Object>(toCompose[2]){

{

addLeftChild(new TerminalPrimitive<Object>(toCompose[1]));

addRightChild(new TerminalPrimitive<Object>(toCompose[3]));

}

});

}

};

case 6:

case 7:

return new NonTerminalPrimitive<Object>(toCompose[1]) {

{

addLeftChild(new TerminalPrimitive<Object>(toCompose[0]));

addRightChild(new NonTerminalPrimitive<Object>(toCompose[3]){

{

addLeftChild(new TerminalPrimitive<Object>(toCompose[2]));

addRightChild(new TerminalPrimitive<Object>(toCompose[4]));

}

});

}

};

case 10:

return new NonTerminalPrimitive<Object>(toCompose[0]);

default:

System.out.println("Error. No such rule");

break;

}

return null;

}

}

package com.lab536.lab4;

import com.lab536.lab2.Operations;

public enum RuleTypes implements Operations{

EXPRESSION,

L\_EXPRESSION,

VARIABLE,

CONSTANT,

OPERATION,

OPER\_LIST;

}

package com.lab536.lab4;

import com.lab536.lab2.AbstractPrimitive;

import com.lab536.lab2.TerminalPrimitive;

import com.lab536.lab3.Token;

public class RuleHandleDecorator<E> extends AbstractPrimitive<E> {

private AbstractPrimitive<?> data;

public final RuleTypes type;

public RuleHandleDecorator(AbstractPrimitive<?> data, RuleTypes type){

if((data != null)) {

// && (content.getClass() == RuleHandleDecorator.class)){

this.data = (AbstractPrimitive<?>)data;

this.type = type;

}

else{

throw new IllegalArgumentException("Some data is null");

}

}

public RuleHandleDecorator(Token t){

if(t != null){

this.data = new TerminalPrimitive<Token>(t);

switch(t.getType()){

case IDENTIFIER:

type = RuleTypes.VARIABLE;

break;

default:

type = RuleTypes.CONSTANT;

break;

}

}

else{

throw new IllegalArgumentException("Some input data is missing");

}

}

@Override

protected void addChild(AbstractPrimitive<?> child, int side) {

throw new IllegalAccessError("This class can't have children");

}

@Override

public String interpret() {

return data.interpret();

}

@Override

public Object getContent() {

return type;

}

}

package com.lab536.lab4.test;

import org.junit.Test;

import com.lab536.lab1.DataTable;

import com.lab536.lab3.Tokenizer;

import com.lab536.lab4.Parser;

import static org.junit.Assert.\*;

public class ParserTest {

@Test

public void testSimple(){

Tokenizer t = new Tokenizer("b := a;");

DataTable dt = t.tokenize();

Parser p = new Parser(dt);

p.parse();

assertTrue(p.getTree().interpret().equals("b:=a;"));

}

@Test

public void testAssign(){

Tokenizer t = new Tokenizer("b := a[n];");

DataTable dt = t.tokenize();

Parser p = new Parser(dt);

p.parse();

assertTrue(p.getTree().interpret().equals("b:=a[n];"));

}

@Test

public void testAssignWArrayOperation(){

Tokenizer t = new Tokenizer("b := a[n] + 42;");

DataTable dt = t.tokenize();

Parser p = new Parser(dt);

p.parse();

assertTrue(p.getTree().interpret().equals("b:=a[n]+42;"));

}

@Test

public void testRepeat(){

Tokenizer t = new Tokenizer("repeat b := a[n]+\"q42\"; until n=0;");

DataTable dt = t.tokenize();

Parser p = new Parser(dt);

p.parse();

assertTrue(p.getTree().interpret().equals("repeatb:=a[n]+\"q42\";untiln=0;"));

}

@Test

public void testRepeatWrong(){

Tokenizer t = new Tokenizer("repeat b := a[n]+\"q42\"; until (n=0);");

DataTable dt = t.tokenize();

Parser p = new Parser(dt);

p.parse();

assertTrue(p.getTree() == null);

}

@Test

public void testRepeatParenthesis(){

Tokenizer t = new Tokenizer("repeat b := a[n]+\"q42\"; until n=(a+2);");

DataTable dt = t.tokenize();

Parser p = new Parser(dt);

p.parse();

assertTrue(p.getTree().interpret().equals("repeatb:=a[n]+\"q42\";untiln=(a+2);"));

}

}