НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ

«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ»

ФАКУЛЬТЕТ ІНФОРМАТИКИ І ОБЧИСЛЮВАЛЬНОЇ ТЕХНІКИ

КАФЕДРА ОБЧИСЛЮВАЛЬНОЇ ТЕХНІКИ

**Лабораторна робота №2**

з дисципліни **«**Системне програмування 2**»**

Виконав:

студент 3 курсу

гр. ІВ-81

Мисак Олександр

Варіант 18

Перевірив:

Павлов В. Г.

Київ 2020 р.

**Тема:** Локальні змінні

**Мета:** побудова компілятору з використанням локальних змінних.

**Завдання:**



**Лістинг програми компілятора:**

**Main.java**

package com.amaterasu.main;  
  
import java.util.Scanner;  
  
public class Main {  
 public static void main(String[] args) {  
 try {  
 LexingChecker lexChecker = new LexingChecker("1-18-Java-IV-81-Mysak.py");  
  
 System.*out*.println("\n====> LEXING CHECK RESULT <====\n");  
 lexChecker.listTokens();  
  
 TokenParser parser = new TokenParser(lexChecker.getTokens());  
 parser.getCoreAST().printAST();  
 System.*out*.println("\n====> END <====\n");  
  
 ASMGenerator asmGen = new ASMGenerator(parser.getCoreAST(), parser.getAstMap());  
 String generatedFileName = "1-18-Java-IV-81-Mysak.asm";  
 boolean success = asmGen.createFile(generatedFileName);  
  
 if (success) {  
 System.*out*.println("Compilation succeeded, " + generatedFileName + " is located in " + System.*getProperty*("user.dir")  
 + "\\" + generatedFileName);  
 } else {  
 System.*err*.println("Compilation failed");  
 }  
 } catch (ParseException e) {  
 System.*err*.println(e.getMessage());  
 } finally {  
 Scanner scan = new Scanner(System.*in*);  
 scan.nextLine();  
 scan.close();  
 }  
 }  
}

**ParseException.java**

package com.amaterasu.main;  
  
public class ParseException extends Exception {  
 private static final long serialVersionUID = 227786097791240244L;  
  
 public ParseException(String message, int row, int column) {  
 super(String.format("%s\n\tat row=%d column=%d", message, (row + 1), (column + 1)));  
 }  
}

**LexingChecker.java**

package com.amaterasu.main;  
  
import org.json.\*;  
  
import java.io.IOException;  
import java.util.ArrayList;  
import java.util.HashMap;  
import java.util.Map;  
import com.fasterxml.jackson.databind.ObjectMapper;  
  
public class LexingChecker {  
 private String text;  
  
 private Map<String, String> keywords;  
 private Map<String, String> symbols;  
 private Map<String, String> whitespaces;  
  
 private final ArrayList<Token> tokens = new ArrayList<>();  
  
 public LexingChecker(String filePath) {  
 keywords = new HashMap<>();  
 symbols = new HashMap<>();  
 whitespaces = new HashMap<>();  
 populateData("3-18-Java-IV-81-Mysak/util/python.json");  
  
 try {  
 text = Util.readFile(filePath);  
 } catch (IOException e) {  
 System.out.println(e.getMessage());  
 }  
  
 generateTokens();  
 }  
  
 private void populateData(String pathToTokens) {  
 try {  
 String fileContents = Util.readFile(pathToTokens);  
 JSONObject pythonTokensJSON = new JSONObject(fileContents);  
 keywords = new ObjectMapper().readValue(pythonTokensJSON.get("keywords").toString(), HashMap.class);  
 symbols = new ObjectMapper().readValue(pythonTokensJSON.get("symbols").toString(), HashMap.class);  
 whitespaces = new ObjectMapper().readValue(pythonTokensJSON.get("whitespaces").toString(), HashMap.class);  
 } catch (IOException e ) {  
 System.out.println(e.getMessage());  
 }  
 }  
  
 private void generateTokens() {  
 tokens.add(new Token("null", "START", -1, -1));  
 if (text.startsWith("null")) {  
 text = text.substring(4);  
 }  
 String[] parseLines = text.split("\n");  
  
 for (int i = 0; i < parseLines.length; i++) {  
 if (parseLine(parseLines[i], i)) {  
 tokens.add(new Token("\n", whitespaces.get("\n"), i, parseLines.length));  
 }  
 }  
 tokens.add(new Token("null", "END", -1, -1));  
 }  
  
 private boolean parseLine(String line, int row) {  
 String cleanLine = line.split("#")[0];  
 if (cleanLine.matches("^\\s\*$")) {  
 return false;  
 }  
  
 String[] splitCodeLine = cleanLine.split("");  
 boolean tabSpace = true;  
  
 for (int i = 0; i < splitCodeLine.length; i++) {  
 if (i + 1 != splitCodeLine.length && symbols.containsKey(splitCodeLine[i] + splitCodeLine[i + 1])) {  
 tokens.add(new Token(splitCodeLine[i] + splitCodeLine[i + 1], symbols.get(splitCodeLine[i] + splitCodeLine[i + 1]), row, i));  
 i++;  
 } else {  
 if (symbols.containsKey(splitCodeLine[i])) {  
 if (i + 1 != splitCodeLine.length && (splitCodeLine[i].matches("[\"']"))) {  
 StringBuilder num = new StringBuilder(splitCodeLine[i]);  
 short j = 1;  
 while (i + j < splitCodeLine.length) {  
 num.append(splitCodeLine[i + j]);  
 j++;  
 if (splitCodeLine[i + j - 1].equals(splitCodeLine[i])) {  
 break;  
 }  
 }  
  
 tokens.add(new Token(num.toString(), "STRING", row, i));  
 i += num.length() - 1;  
 } else {  
 tokens.add(new Token(splitCodeLine[i], symbols.get(splitCodeLine[i]), row, i));  
 }  
 } else {  
 if (whitespaces.containsKey(splitCodeLine[i])) {  
 if (tabSpace) {  
 tokens.add(new Token(splitCodeLine[i], whitespaces.get(splitCodeLine[i]), row, i));  
 }  
 } else {  
 if (i + 1 != splitCodeLine.length && splitCodeLine[i].equals("0") && splitCodeLine[i + 1].matches("[xob]")) {  
 StringBuilder nonDecimalNum = new StringBuilder("0" + splitCodeLine[i + 1]);  
 short j = 2;  
 while ((i + j < splitCodeLine.length) && isValidNum(splitCodeLine[i + j], splitCodeLine[i + 1]) && j < 8) {  
 nonDecimalNum.append(splitCodeLine[i + j]);  
 j++;  
 }  
  
 String numType = "HEX";  
 switch (splitCodeLine[i + 1]) {  
 case "o":  
 numType = "OCT";  
 break;  
 case "b":  
 numType = "BIN";  
 break;  
 default:  
 break;  
 }  
 tokens.add(new Token(nonDecimalNum.toString(), numType, row, i));  
  
 i += nonDecimalNum.length() - 1;  
 } else {  
 if (splitCodeLine[i].matches("\\d")) {  
 StringBuilder stringifiedNumber = new StringBuilder();  
 boolean isFloat = false;  
 short j = 0;  
 while (i + j < splitCodeLine.length && splitCodeLine[i + j].matches("[\\d.]")) {  
 if (splitCodeLine[i + j].equals(".")) {  
 isFloat = true;  
 }  
 stringifiedNumber.append(splitCodeLine[i + j]);  
 j++;  
 }  
  
 if (isFloat) {  
 tokens.add(new Token(stringifiedNumber.toString(), "FLOAT", row, i));  
 } else {  
 tokens.add(new Token(stringifiedNumber.toString(), "INT", row, i));  
 }  
 i += stringifiedNumber.length() - 1;  
 } else {  
 if (splitCodeLine[i].matches("[a-zA-Z]")) {  
 StringBuilder stringifiedNumber = new StringBuilder();  
 short j = 0;  
 while (i + j < splitCodeLine.length && splitCodeLine[i + j].matches("\\w")) {  
 stringifiedNumber.append(splitCodeLine[i + j]);  
 j++;  
 }  
  
 tokens.add(new Token(stringifiedNumber.toString(), keywords.getOrDefault(stringifiedNumber.toString(), "IDENTIFIER"), row, i));  
 tabSpace = false;  
 i += stringifiedNumber.length() - 1;  
 } else {  
 tokens.add(new Token(splitCodeLine[i], "UNDEF", row, i));  
 try {  
 throw new ParseException("Undefined symbol", row, i);  
 } catch (ParseException e) {  
 System.err.println(e.getMessage());  
 System.err.println((int) splitCodeLine[i].toCharArray()[0]);  
 }  
 }  
 }  
 }  
 }  
 }  
 }  
 }  
 return true;  
 }  
  
 private boolean isValidNum(String s, String system) {  
 switch (system) {  
 case "x":  
 return s.matches("[\\da-fA-F]");  
 case "o":  
 return s.matches("[0-7]");  
 case "b":  
 return s.equals("1") || s.equals("0");  
 default:  
 return false;  
 }  
 }  
   
 public ArrayList<Token> getTokens() {  
 return tokens;  
 }  
  
 public void listTokens() {  
 for (Token token : tokens) {  
 String tokenValue = token.getValue();  
 switch (tokenValue) {  
 case "\n":  
 tokenValue = "\\n";  
 break;  
 case "\t":  
 tokenValue = "\\t";  
 break;  
 case " ":  
 tokenValue = "\\s";  
 break;  
 default:  
 break;  
 }  
 System.out.printf("%s => %s%n", tokenValue, token.getType());  
 }  
 }  
}

**TokenParser.java**

package com.amaterasu.main;  
  
import org.json.JSONArray;  
import org.json.JSONObject;  
  
import java.io.IOException;  
import java.util.\*;  
  
  
public class TokenParser {  
 private final HashMap<String, AST> astMap;  
 private final AST coreAST;  
 private final HashMap<String, String[]> templates;  
 private final HashMap<Integer, String> priorities;  
  
 public TokenParser(ArrayList<Token> tokens) throws ParseException {  
 TwoWayIterator<Token> tokenIterator = new TwoWayIterator<>(tokens);  
  
 this.astMap = new HashMap<>();  
 this.coreAST = new AST(tokenIterator.next());  
 this.templates = new HashMap<>();  
 this.priorities = new HashMap<>();  
  
 populateTemplates("3-18-Java-IV-81-Mysak/util/templates.json");  
 populatePriorities("");  
 run(tokenIterator);  
 }  
  
 private void populateTemplates(String pathToTemplates) {  
 try {  
 String fileContents = Util.*readFile*(pathToTemplates);  
 JSONObject templatesJSON = new JSONObject(fileContents);  
 Iterator<String> keysIterator = templatesJSON.keys();  
 while (keysIterator.hasNext()) {  
 String key = keysIterator.next();  
 JSONArray templateArray = templatesJSON.getJSONArray(key);  
 ArrayList<String> templateList = new ArrayList<>();  
 for (int i = 0; i < templateArray.length(); i++) {  
 templateList.add(templateArray.getString(i));  
 }  
 String[] templateStringArray = templateList.toArray(new String[0]);  
 templates.put(key, templateStringArray);  
 }  
 } catch (IOException e) {  
 System.*out*.println(e.getMessage());  
 }  
 }  
  
 private void populatePriorities(String pathToPriorities) {  
 priorities.put(4, "POW");  
 priorities.put(5, "(ADD)|(SUB)|(BIT\_NOT)");  
 priorities.put(6, "(MUL)|(DIV)|(INT\_DIV)|(PERCENT)");  
 priorities.put(7, "(ADD)|(SUB)");  
 priorities.put(8, "(L\_SHIFT)|(R\_SHIFT)");  
 priorities.put(9, "BIT\_AND");  
 priorities.put(10, "BIT\_XOR");  
 priorities.put(11, "BIT\_OR");  
 priorities.put(12, "(IN)|(IS)|(LT)|(GT)|(LE)|(GE)|(NE)|(EQ)");  
 priorities.put(13, "NOT");  
 priorities.put(14, "AND");  
 priorities.put(15, "OR");  
 }  
  
 private void run(TwoWayIterator<Token> tokenIterator) throws ParseException {  
 while (tokenIterator.next().getType().equals("DECLARATION")) {  
 tokenIterator.prev();  
 AST tmp = new AST(parseDefinition(tokenIterator));  
 astMap.put(tmp.getRoot().getCurrent().getValue(), tmp);  
 }  
 tokenIterator.prev();  
 while (tokenIterator.hasNext()) {  
 if (tokenIterator.next().getType().equals("END")) {  
 break;  
 }  
 tokenIterator.prev();  
 ASTNode call = parseDefinitionCall(tokenIterator);  
 if (!astMap.containsKey(call.getCurrent().getValue()))  
 throwAndPrint("Unexpected call", call.getCurrent());  
 call.appendChild(astMap.get(call.getCurrent().getValue()).getRoot());  
 coreAST.getRoot().appendChild(call);  
 }  
 }  
  
 private ASTNode parseDefinition(TwoWayIterator<Token> tokenIterator) throws ParseException {  
 int spaceCount = -1;  
 Token declarationName = null;  
 Token token = null;  
 ArrayList<ASTNode> statements;  
  
 for (String part : templates.get("FUNC")) {  
 token = tokenIterator.next();  
 switch (part) {  
 case "BLANK": {  
 if (!token.getType().matches("(TAB)|(SPACE)")) {  
 System.*out*.println(token.getType());  
 System.*out*.println("96");  
 throwAndPrint("Incorrect type", token);  
 }  
 break;  
 }  
 case "IDENTIFIER": {  
 declarationName = token;  
 break;  
 }  
 default: {  
 if (!token.getType().equals(part)) {  
 if (part.equals("RBR")) {  
 throwAndPrint(String.*format*("Expected closing ')'. Found: '%s'", token.getValue()), token);  
 }  
 System.*out*.println(part);  
 System.*out*.println(token.getType());  
 System.*out*.println("111");  
 throwAndPrint("Incorrect type", token);  
 }  
 }  
 }  
 }  
  
 tokenIterator.prev();  
 statements = new ArrayList<>();  
  
 assert token != null;  
 assert declarationName != null;  
  
 while (token.getType().equals("TAB") || token.getType().equals("SPACE")) {  
 int tempSpaces = 0;  
 tokenIterator.next();  
  
 while (token.getType().matches("(TAB)|(SPACE)")) {  
 if (token.getType().equals("TAB"))  
 tempSpaces += 8;  
 else  
 tempSpaces++;  
 token = tokenIterator.next();  
 }  
  
 if (spaceCount == -1)  
 spaceCount = tempSpaces;  
 if (tempSpaces != spaceCount) {  
 throwAndPrint("Incorrect tab count", token);  
 }  
 tokenIterator.prev();  
  
 statements.add(parseStat(tokenIterator));  
  
 token = tokenIterator.next();  
 tokenIterator.prev();  
 }  
 ASTNode def = new ASTNode(new Token(declarationName.getValue(), "DEF\_IDENTIFIER",  
 declarationName.getRow(), declarationName.getColumn()));  
  
 def.appendChildren(statements);  
 for (ASTNode child : statements) {  
 child.setParent(def);  
 }  
  
 return def;  
 }  
  
 private ASTNode parseStat(TwoWayIterator<Token> tokenIterator) throws ParseException {  
 Token token;  
  
 token = tokenIterator.next();  
 tokenIterator.prev();  
 if (token.getType().equals("RETURN")) {  
 tokenIterator.next();  
 ASTNode returnNode = new ASTNode(new Token("return", "RETURN",  
 token.getRow(), token.getColumn())),  
 retExp = parseSlug(tokenIterator);  
  
 returnNode.appendChild(retExp);  
 retExp.setParent(returnNode);  
  
 token = tokenIterator.next();  
 if (!token.getType().equals("NEW\_LINE")) {  
 System.*out*.println(token.getType());  
 System.*out*.println("176");  
 throwAndPrint("Incorrect type", token);  
 }  
  
 return returnNode;  
 } else {  
 ASTNode exp = parseSlug(tokenIterator);  
  
 token = tokenIterator.next();  
 if (!token.getType().equals("NEW\_LINE")) {  
 System.*out*.println(token.getType());  
 System.*out*.println("187");  
 throwAndPrint("Incorrect type", token);  
 }  
  
 return exp;  
 }  
 }  
  
 enum ExpressionType {  
 *EXP*, *TERM* }  
  
 // Parses either expression or term  
 private ASTNode parseSlug(TwoWayIterator<Token> tokenIterator) throws ParseException {  
 Token token = tokenIterator.next(),  
 token2 = tokenIterator.next();  
  
 tokenIterator.prev();  
 tokenIterator.prev();  
 if (token.getType().equals("IDENTIFIER") && token2.getType().equals("ASSIGNMENT")) {  
 tokenIterator.next();  
 tokenIterator.next();  
 ASTNode id = new ASTNode(new Token(token.getValue(), "ID", token.getRow(), token.getColumn()));  
  
 ASTNode assign = new ASTNode(token2),  
 exp = parseSlug(tokenIterator);  
  
 assign.appendChild(exp);  
 exp.setParent(assign);  
  
 id.appendChild(assign);  
 assign.setParent(id);  
  
 return id;  
 } else {  
 return parsePriority(15, tokenIterator);  
 }  
 }  
  
 private ASTNode parsePriority(int prior, TwoWayIterator<Token> tokenIterator) throws ParseException {  
 ASTNode operationSign = null, left;  
 if (prior <= 6)  
 left = parseFactor(tokenIterator);  
 else  
 left = parsePriority(prior - 1, tokenIterator);  
  
 ArrayList<ASTNode> nodeQueue = new ArrayList<>(),  
 operationQueue = new ArrayList<>();  
 nodeQueue.add(left);  
  
 Token token = tokenIterator.next();  
 tokenIterator.prev();  
  
 if (token.getType().matches(priorities.get(prior))) {  
 while (token.getType().matches(priorities.get(prior))) {  
 tokenIterator.next();  
  
 operationQueue.add(new ASTNode(token));  
  
 ASTNode node;  
 if (prior <= 6)  
 node = parseFactor(tokenIterator);  
 else  
 node = parsePriority(prior - 1, tokenIterator);  
 nodeQueue.add(node);  
  
 token = tokenIterator.next();  
 tokenIterator.prev();  
 }  
 Collections.*reverse*(operationQueue);  
 Collections.*reverse*(nodeQueue);  
  
 TwoWayIterator<ASTNode> operationIterator = new TwoWayIterator<>(operationQueue);  
 TwoWayIterator<ASTNode> nodeIterator = new TwoWayIterator<>(nodeQueue);  
 operationSign = operationIterator.next();  
 operationSign.appendChild(nodeIterator.next());  
  
 while (operationIterator.hasNext()) {  
 ASTNode deepest = operationSign.getDeepestLeft(),  
 tmpOper = operationIterator.next(),  
 tmpNode = nodeIterator.next();  
  
 tmpOper.appendChild(tmpNode);  
 tmpNode.setParent(tmpOper);  
  
 deepest.setFirstChild(tmpOper);  
 tmpOper.setParent(deepest);  
 }  
 operationSign.getDeepestLeft().setFirstChild(nodeIterator.next());  
 }  
 return operationSign == null ? left : operationSign;  
 }  
  
 private ASTNode parseFactor(TwoWayIterator<Token> tokenIterator) throws ParseException {  
 Token token = tokenIterator.next();  
  
 ASTNode operation;  
 if (token.getType().equals("LBR")) {  
 ASTNode exp = parseSlug(tokenIterator);  
 if (!tokenIterator.next().getType().equals("RBR")) {  
 throwAndPrint(String.*format*("Expected closing ')'. Found: %s", tokenIterator.current().getValue()), tokenIterator.current());  
 }  
 return exp;  
 } else {  
 if (token.getType().matches("(ADD)|(SUB)|(NOT)")) {  
 operation = new ASTNode(new Token(token.getValue(), "UNARY\_" + token.getType(),  
 token.getRow(), token.getColumn()));  
 ASTNode nextTerm = parseFactor(tokenIterator);  
 operation.appendChild(nextTerm);  
 assert nextTerm != null;  
 nextTerm.setParent(operation);  
 return operation;  
 } else {  
 if (token.getType().matches("(INT)|(FLOAT)|(BIN)|(OCT)|(HEX)|(STRING)")) {  
 return parseExpression(token);  
 } else {  
 throwAndPrint("Unexpected token", token);  
 }  
 }  
 }  
 return null;  
 }  
  
 private ASTNode parseExpression(Token token) throws ParseException {  
 String value = token.getValue();  
  
 switch (token.getType()) {  
 case "INT": {  
 return new ASTNode(token);  
 }  
 case "FLOAT": {  
 StringBuilder casted = new StringBuilder();  
 for (char ch : value.toCharArray()) {  
 if (ch == '.') {  
 break;  
 }  
 casted.append(ch);  
 }  
 return new ASTNode(new Token(casted.toString(),  
 "INT(FLOAT)", token.getRow(), token.getColumn()));  
 }  
 case "HEX": {  
 return new ASTNode(new Token(Long.*decode*(value).toString(),  
 "INT(HEX)", token.getRow(), token.getColumn()));  
 }  
 case "OCT": {  
 return new ASTNode(new Token(Integer.*parseInt*(value.substring(2), 8) + "",  
 "INT(OCT)", token.getRow(), token.getColumn()));  
 }  
 case "BIN": {  
 return new ASTNode(new Token(Integer.*parseInt*(value.substring(2), 2) + "",  
 "INT(BIN)", token.getRow(), token.getColumn()));  
 }  
 case "STRING": {  
 if (value.length() == 3) {  
 return new ASTNode(new Token((int) value.toCharArray()[1] + "",  
 "INT(CHAR)", token.getRow(), token.getColumn()));  
 } else {  
 throwAndPrint("Casting error", token);  
 }  
 break;  
 }  
 }  
 return null;  
 }  
  
 private ASTNode parseDefinitionCall(TwoWayIterator<Token> tokenIterator) throws ParseException {  
 Token token;  
 ASTNode defCall = null;  
 for (String lexeme : templates.get("CALL")) {  
 token = tokenIterator.next();  
 if (lexeme.equals("IDENTIFIER")) {  
 defCall = new ASTNode(new Token(token.getValue(), "CALL",  
 token.getRow(), token.getColumn()));  
 } else {  
 if (!token.getType().equals(lexeme)) {  
 throwAndPrint("Unexpected token", token);  
 }  
 }  
 }  
  
 return defCall;  
 }  
  
 private void throwAndPrint(String message, Token token) throws ParseException {  
 throw new ParseException(message, token.getRow(), token.getColumn());  
 }  
  
  
 public HashMap<String, AST> getAstMap() {  
 return astMap;  
 }  
  
 public AST getCoreAST() {  
 return coreAST;  
 }  
}

**AST.java**

package com.amaterasu.main;  
  
public class AST {  
 private final ASTNode root;  
  
 public AST(ASTNode root) {  
 this.root = root;  
 }  
  
 public AST(Token token) {  
 this.root = new ASTNode(token, null);  
 }  
  
 public void printAST() {  
 System.out.println("\n====> SYNTAX TREE <====\n");  
 printChildren(root, 0);  
 }  
  
 private void printChildren(ASTNode child, int depth) {  
 StringBuilder prev = new StringBuilder();  
 if (depth > 1) {  
 for (int i = 0; i < depth - 1; i++) {  
 prev.append("\t");  
 }  
 prev.append("=> ");  
 } else {  
 if (depth > 0) {  
 prev = new StringBuilder("=> ");  
 }  
 }  
 System.out  
 .printf("%s%1s (%-1s)%n", prev.toString(), child.getCurrent().getValue(), child.getCurrent().getType());  
 for (ASTNode node : child.getChildren()) {  
 printChildren(node, depth + 1);  
 }  
 }  
  
 public ASTNode getRoot() {  
 return root;  
 }  
}

**ASTNode.java**

package com.amaterasu.main;  
  
import java.util.ArrayList;  
  
public class ASTNode {  
 private final Token current;  
 private ASTNode parent;  
 private final ArrayList<ASTNode> children;  
  
 public ASTNode(Token token){  
 this.current = token;  
 this.parent = null;  
 this.children = new ArrayList<>();  
 }  
  
 public ASTNode(Token token, ASTNode parent){  
 this.current = token;  
 this.parent = parent;  
 this.children = new ArrayList<>();  
 }  
  
 public ASTNode(Token token, ASTNode parent, ArrayList<ASTNode> children){  
 this.current = token;  
 this.parent = parent;  
 this.children = children;  
 }  
  
 public ASTNode getParent() {  
 return parent;  
 }  
  
 public Token getCurrent() {  
 return current;  
 }  
  
 public ArrayList<ASTNode> getChildren() {  
 return children;  
 }  
  
 public ASTNode getChild(int id){  
 return children.get(id);  
 }  
  
 public ASTNode getDeepestLeft(){  
 return goDeeperLeft(this);  
 }  
  
 private ASTNode goDeeperLeft(ASTNode start){  
 if (start.getChildren().size() > 1)  
 return goDeeperLeft(start.getChild(0));  
 else  
 return start;  
 }  
  
 public ASTNode getDeepestRight(){  
 return goDeeperRight(this);  
 }  
  
 private ASTNode goDeeperRight(ASTNode start){  
 if (start.getChildren().size() > 1)  
 return goDeeperRight(start.getChild(1));  
 else  
 return start;  
 }  
  
 public ASTNode getDeepestSolo(){  
 return goDeeperSolo(this);  
 }  
  
 private ASTNode goDeeperSolo(ASTNode start){  
 if (start.getChildren().size() > 0)  
 return goDeeperSolo(start.getChild(0));  
 else  
 return start;  
 }  
  
 public void setParent(ASTNode parent) {  
 this.parent = parent;  
 }  
  
 public void insertChild(int position, ASTNode child) {  
 children.add(position, child);  
 }  
  
 public void appendChild(ASTNode child){  
 children.add(child);  
 }  
  
 public void appendChildren(ArrayList<ASTNode> children){ this.children.addAll(children); }  
  
 public void setFirstChild(ASTNode child){  
 insertChild(0, child);  
 }  
}

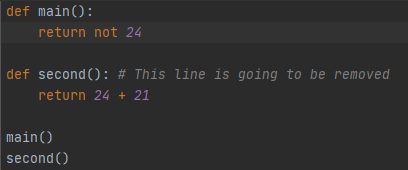
**TwoWayIterator.java (самописний розширений ітератор для можливості зміщати курсор в обидві сторони)**

package com.amaterasu.main;  
  
import java.util.ArrayList;  
import java.util.Iterator;  
  
public class TwoWayIterator<E> implements Iterator<E> {  
 private final ArrayList<E> iterable;  
 private int currentIndex = -1;  
  
 public TwoWayIterator(ArrayList<E> list) {  
 this.iterable = list;  
 }  
  
 @Override  
 public E next() {  
 currentIndex++;  
 return current();  
 }  
  
 public void prev() {  
 currentIndex--;  
 current();  
 }  
  
 @Override  
 public boolean hasNext() {  
 return currentIndex < iterable.size() - 1;  
 }  
  
 @Override  
 public void remove() {  
 iterable.remove(currentIndex);  
 }  
  
 public E current() {  
 return iterable.get(currentIndex);  
 }  
  
 public E get(int i) {  
 return iterable.get(i);  
 }  
}

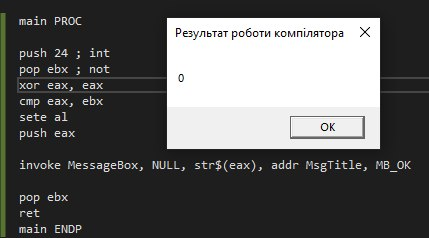
**Util.java (файл з утилітою зчитки з файлу, потенційно буде зберігати в собі дублюючі ділянки коду)**

package com.amaterasu.main;  
  
import java.io.File;  
import java.io.FileReader;  
import java.io.IOException;  
  
public class Util {  
 public static String readFile(String filePath) throws IOException {  
 if (!new File(filePath).exists()) {  
 throw new IOException("File does not exist");  
 }  
 StringBuilder returnText = new StringBuilder();  
 try (FileReader reader = new FileReader(filePath)) {  
 int charCode;  
 while ((charCode = reader.read()) != -1) {  
 returnText.append((char) charCode);  
 }  
 } catch (IOException e) {  
 System.*out*.println(e.getMessage());  
 }  
 return returnText.toString();  
 }  
}

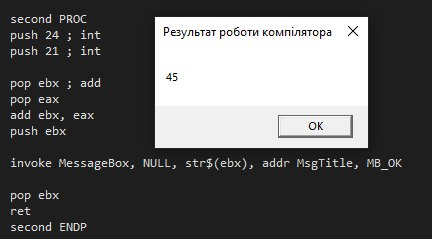
**Попереднi приклади:**



not 24:

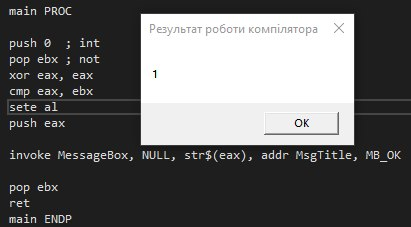


24 + 21:

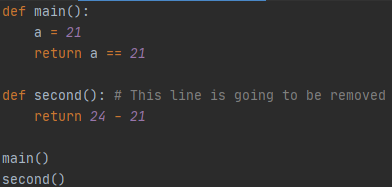


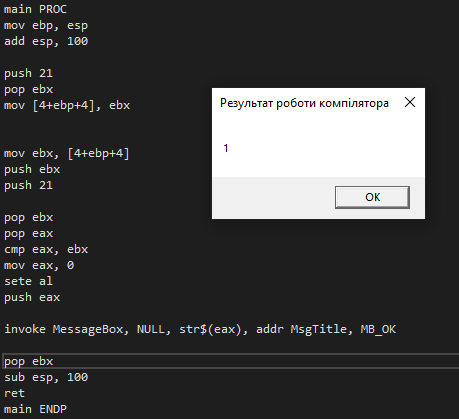
not 0:

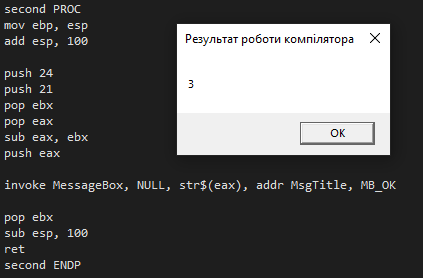




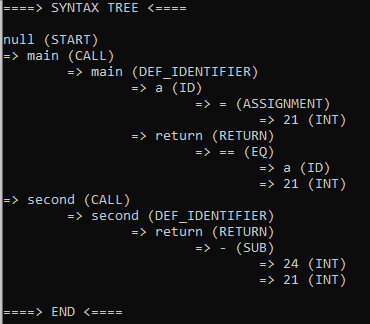
**Новi приклади:**

****





Побудоване дерево AS:



**Контрольні питання**

2. Чи вірний цей вираз: a = 2 \* (b = 2)? Обґрунтуйте свою відповідь.

3. Чому присвоювання не обробляється як звичайний бінарний оператор?

4. Поясніть порядок розташування операторів в граматиці.

5. Навіщо потрібен стек, стекові кадри та пари «ім’я – адреса»? Яка інша назва цих пар?

6. Яку роль виконує регістр EBP?

7. Що таке пролог та епілог функції?

8. Як змінюється індекс стеку?

9. Який алгоритм використання таблиці змінних?

10. Як обробляється пріоритетність?

11. Чому AST залишилось незмінним?

12. Що означає «встановити біт» (set a bit)?

13. Чому використовується прапорець SF?

14. Що таке short-circuit evaluation і чому це важливо чи не важливо?

15. В чому різниця між je та jmp ?

Q: Яка основна відмінність обробки функцій в цій лабораторній роботі?

A: В цій лабораторній роботі була додана обробка локальних змінних.

Q: Чи вірний цей вираз: a = 2 \* (b = 2)? Обґрунтуйте свою відповідь.

A: Даний вираз не є правильним, оскільки змінна b не може ініціалізуватися під час виконання дій з нею.

Q: Чому присвоювання не обробляється як звичайний бінарний оператор?

A: Ліва і права частини бінарного оператору можуть бути будь-якими виразами, а от ліва частина оператора присвоювання – ні.

Q: Поясніть порядок розташування операторів в граматиці

A: Операції в дужках, унарні, інші операції

Q: Навіщо потрібен стек, стекові кадри та пари «ім’я – адреса»? Яка інша назва цих пар?

A:Стек потрібен для зберігання змінних. Пари «ім’я - адреса» потрібні для того щоби знати точне розташування змінної у стеку. Кадр стеку - область пам’яті, що виділяється на вершині стеку під час виклику функції.

Q: Яку роль виконує регістр EBP?

A: Extended Base Pointer - вказівник на початок та кінець стеку.

Q: Що таке пролог та епілог функції?

A: Пролог – створення функцією для себе нового кадру стеку. Епілог – видалення цього кадру стеку.

Q: Як змінюється індекс стеку?

A: Він відповідає за наступне вільне місце в стеку, зсув відносно EBP.

Q: Який алгоритм використання таблиці змінних?

A:

Якщо змінна вже є у таблиці – помилка

Інакше - обчислення виразу та переміщення результату в регістр

Запис змінної в таблицю

Переміщення адреси нижче

Q: Як обробляється пріоритетність?

A: Відповідно правилам граматики.

Q: Чому AST залишилось незмінним?

A: Тому що структура парсеру не змінилася.

Q: Що означає «встановити біт» (set a bit)?

A: Встановлення значення регістру в 1 байт.

Q: Чому використовується прапорець SF?

A: Для визначення більшого операнду в операції порівняння.

Q: Що таке short-circuit evaluation і чому це важливо чи не важливо?

A: Обчислення за скороченою схемою (short-circuit evaluation) полягає в тому, що якщо відомий результат після обчислення першої частини, друга частина рахуватись не буде.

Q: В чому різниця між je та jmp?

A: je – умовний перехід, jmp – безумовний перехід.