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:** обробка унарних та бінарних операцій в компіляторі.

**Мета:** розширення функціоналу компілятора за рахунок реалізації обробки унарних та бінарних операцій.

**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("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;  
  
 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<>();  
  
 populateTemplates("templates.json");  
 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 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[] spacesAndTabs = {0, 0};  
 String declarationSlug = "";  
 Token token;  
 ASTNode declaration = null;  
  
 for (String part : templates.get("FUNC")) {  
 token = tokenIterator.next();  
 switch (part) {  
 case "BLANK": {  
 while (token.getType().matches("(TAB)|(SPACE)")) {  
 if (token.getType().equals("TAB")) {  
 spacesAndTabs[1]++;  
 } else {  
 spacesAndTabs[0]++;  
 }  
 token = tokenIterator.next();  
 }  
 if (spacesAndTabs[0] + spacesAndTabs[1] == 0) {  
 throwAndPrint("Incorrect tab count", token);  
 }  
 break;  
 }  
 case "STAT": {  
 tokenIterator.prev();  
 ASTNode statement = parseStat(tokenIterator);  
 declaration = new ASTNode(new Token(declarationSlug, "DECLARATION\_NAME",  
 token.getRow(), token.getColumn()));  
 declaration.appendChild(statement);  
 statement.setParent(declaration);  
 break;  
 }  
 case "IDENTIFIER": {  
 declarationSlug = token.getValue();  
 break;  
 }  
 default: {  
 if (!token.getType().equals(part)) {  
 if (part.equals("RBR")) {  
 throwAndPrint(String.*format*("Expected closing ')'. Found: '%s'", token.getValue()), token);  
 }  
 throwAndPrint("Incorrect type", token);  
 }  
 }  
 }  
 }  
 return declaration;  
 }  
  
 private ASTNode parseStat(TwoWayIterator<Token> tokenIterator) throws ParseException {  
 Token token;  
 ASTNode statement = null;  
 tokenIterator.prev();  
 for (String part : templates.get("STAT")) {  
 token = tokenIterator.next();  
 if (part.equals("EXP")) {  
 tokenIterator.prev();  
 ASTNode exp = parseSlug(ExpressionType.*EXP*, tokenIterator);  
 statement = new ASTNode(new Token("return", "RETURN",  
 token.getRow(), token.getColumn()));  
 statement.appendChild(exp);  
 exp.setParent(statement);  
 } else {  
 if (!token.getType().equals(part)) {  
 throwAndPrint("Incorrect type", token);  
 }  
 }  
 }  
 return statement;  
 }  
  
 enum ExpressionType {  
 *EXP*, *TERM* }

// Parses either expression or term  
 private ASTNode parseSlug(ExpressionType type, TwoWayIterator<Token> tokenIterator) throws ParseException {  
 ASTNode slug = type == ExpressionType.*EXP* ? parseSlug(ExpressionType.*TERM*, tokenIterator) : parseFactor(tokenIterator),  
 operation = null;  
 Token token = tokenIterator.next();  
 tokenIterator.prev();  
 String regToMatch = type == ExpressionType.*EXP* ? "(ADD)|(SUB)" : "(MUL)|(DIV)|(INT\_DIV)";  
 while (token.getType().matches(regToMatch)) {  
 tokenIterator.next();  
 operation = new ASTNode(token);  
 ASTNode nextTerm = type == ExpressionType.*EXP* ? parseSlug(ExpressionType.*TERM*, tokenIterator) : parseFactor(tokenIterator);  
 operation.appendChild(slug);  
 operation.appendChild(nextTerm);  
 assert slug != null;  
 assert nextTerm != null;  
 slug.setParent(operation);  
 nextTerm.setParent(operation);  
 token = tokenIterator.next();  
 tokenIterator.prev();  
 }  
  
 return operation == null ? slug : operation;  
 }  
  
 private ASTNode parseFactor(TwoWayIterator<Token> tokenIterator) throws ParseException {  
 Token token = tokenIterator.next();  
  
 ASTNode operation;  
 if (token.getType().equals("LBR")) {  
 ASTNode exp = parseSlug(ExpressionType.*EXP*, 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();  
 String type = token.getType();  
  
 switch (token.getType()) {  
 case "INT": {  
 break;  
 }  
 case "FLOAT": {  
 StringBuilder casted = new StringBuilder();  
 for (char ch : value.toCharArray()) {  
 if (ch == '.') {  
 break;  
 }  
 casted.append(ch);  
 }  
 value = casted.toString();  
 type = "INT(FLOAT)";  
 break;  
 }  
 case "HEX":  
 case "OCT":  
 case "BIN": {  
 try {  
 value = Long.*decode*(value).toString();  
 } catch (Exception e) {  
 value = Integer.*valueOf*(value.substring(2), 2).toString();  
 }  
 break;  
 }  
 case "STRING": {  
 if (value.length() == 3) {  
 value = (int) value.toCharArray()[1] + "";  
 type = "INT(CHAR)";  
 } else {  
 throwAndPrint("Casting error", token);  
 }  
 break;  
 }  
 default:  
 return null;  
 }  
 return new ASTNode(new Token(value, type, token.getRow(), token.getColumn()));  
 }  
  
 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 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);  
 }  
}

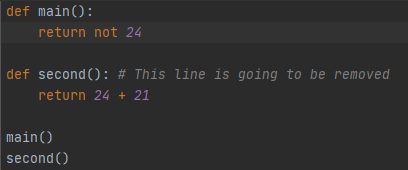
**TwoWayIterator.java (самописний розширений ітератор для можливості зміщати курсор в обидві сторони)**

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);  
 }  
}

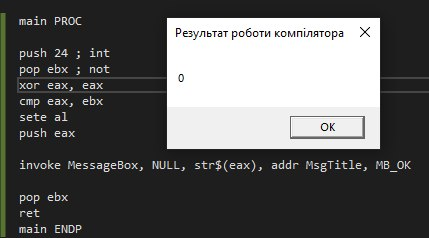
**Util.java (файл з утилітою зчитки з файлу, потенційно буде зберігати в собі дублюючі ділянки коду)**

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();  
 }  
}

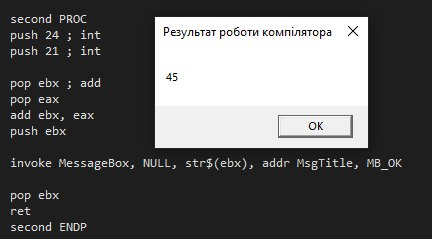
**Контрольнi приклади:**



not 24:

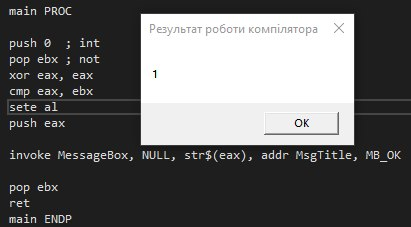


24 + 21:



not 0:

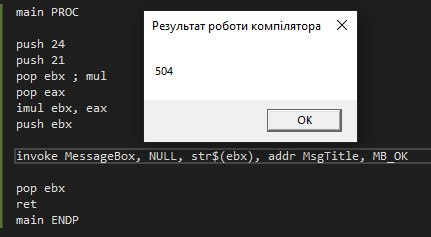




Було також реалізовано компіляцію результату інших унарних операторів (власне, майже всіх) та арифметичних дій.

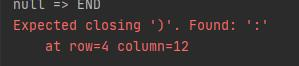
Наприклад:



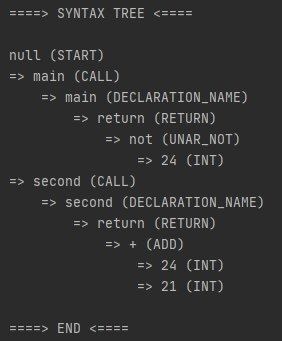


Синтаксичні помилки:





Побудоване дерево AS:



**Контрольні питання**

Q: В чому суть Incremental Approach і як він використовується в цій роботі?

A: В тому, щоб поступово ускладнювати логiку продукту задля простiшого досягнення кiнцевої мети. Також цей підхід допоможе не загубитись у великих об’ємах написаного коду, просто його підтримувати та продовжувати розробку. В цій лабораторній роботі в проект було додано можливість розпізнавати унарні та бінарні оператори, це нова функція, яка ускладнює логіку компілятора і таким чином модифікує його у кращу сторону. Це і є IA.

Q: Чому використовується тільки одна лексема «-», хоча є 2 різні операції з цією лексемою?

A: Тому що компілятор сам матиме можливість розрізняти коли ця лексема використовується як бінарний оператор і навпаки, тому що парсер буде перевіряти на присутність двох констант.

Q: В чому різниця між expressions та statements?

A: **Expression** – це будь-яка допустима комбінація констант, змінних, операторів та викликів функції, що може бути представлена значенням якогось типу, наприклад, **int**.

**Statements** модифікують змінну, або ж мають якісь інші «побічні» ефекти, під час обрахунку **expression**.

Q: Як саме парсер визначає наступну операцію?

A: За допомогою LL граматики.

Q: Що таке ліво-рекурсивна граматика і в чому її недоліки?

A: Приблизно так виглядає ліво-рекурсивна граматика:

expr: expr '+' term | term

Її недолік у тому що парсер з такою граматикою буде просто падати через переповнення стеку. (Нескінченний цикл)

Q: Як вирішується проблема пріоритетності операцій та асоціативності?

A: Розширеною формою Бекуса-Наура:

**<exp> ::= <term> { ("+" | "-") <term> }  
<term> ::= <factor> { ("\*" | "/") <factor> }  
<factor> ::= "(" <exp> ")" | <unary\_op> <factor> | <int>**

Q: Опишіть концепцію процесу.

A: Ця форма передбачає те, що term’ів мож бути декілька, це і є її основна перевага.

Q: Призначення регістру ESP та особливості роботи з ним.

A: Це регістр, який зберігає адресу вершини стеку. Інструкція push – покласти значення в стек, а pop – дістати значення зі стеку. При виконанні команди push зміст регістру ESP зменшується, а при виконанні pop - збільшується.

Q: Як виконується операція ділення на мові асемблера?

A: idiv <reg> сприймає EDX та EAX як єдиний 64-бітний регістр і рахує [EDX:EAX] / <reg>. Потім зберігає частку в EAX, остачу в EDX. Тому спочатку необхідно помістити е1 в EАX і розширити його знак в EDX користуючись командою cdq. Вже після цього використовується idiv.