НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ

«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ»

ФАКУЛЬТЕТ ІНФОРМАТИКИ І ОБЧИСЛЮВАЛЬНОЇ ТЕХНІКИ

КАФЕДРА ОБЧИСЛЮВАЛЬНОЇ ТЕХНІКИ

**Лабораторна робота №1**

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

Виконав:

студент 3 курсу

гр. ІВ-81

Мисак Олександр

Варіант 18

Перевірив:

Павлов В. Г.

Київ 2020 р.

**Тема:** розробка та опрацювання базової структури компілятора

**Мета:** знайомство і опанування основних етапів компілювання, а також створення найпростішого компілятору.

**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.File;

import java.io.FileReader;

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 = readFile(filePath);

    } catch (IOException e) {

      System.out.println(e.getMessage());

    }

    generateTokens();

  }

  private 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();

  }

  private void populateData(String pathToTokens) {

    try {

      String fileContents = 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() {

    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));

      }

    }

  }

  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 java.util.ArrayList;

import java.util.Arrays;

import java.util.HashMap;

import java.util.Iterator;

public class TokenParser {

  private final HashMap<String, AST> astMap;

  private final AST coreAST;

  public TokenParser(ArrayList<Token> tokens) throws ParseException {

    this.astMap = new HashMap<>();

    this.coreAST = new AST(new Token(null, "BEGIN", 0, 0));

    for (Iterator<Token> tokenIterator = tokens.iterator(); tokenIterator.hasNext();) {

      Token token = tokenIterator.next();

      String tokenValue = token.getValue();

      String tokenType = token.getType();

      if (tokenType.equals("DECLARATION")) {

        AST astBranch = new AST(parseDefinition(token, tokenIterator));

        astMap.put(astBranch.getRoot().getCurrent().getValue(), astBranch);

      } else if (tokenType.equals("IDENTIFIER")) {

        coreAST.getRoot().appendChild(parseIdentifier(token, tokenIterator));

      } else {

        System.out.println(tokenValue + " " + tokenType);

      }

    }

  }

  private void verifyLiteral(Token token, String expectedLiteral) throws ParseException {

    if (!token.getType().matches(expectedLiteral)) {

      throwAndPrint("Invalid type", token);

    }

  }

  private void iterateAndVerify(Iterator<Token> tokenIterator, String[] expectedTypes) throws ParseException {

    for (String expectedType : expectedTypes) {

      Token token = tokenIterator.next();

      verifyLiteral(token, expectedType);

    }

  }

  private ASTNode parseDefinition(Token prev, Iterator<Token> tokenIterator) throws ParseException {

    int[] spacesAndTabs = { 0, 0 };

    Token token = tokenIterator.next();

    String declarationName = token.getValue();

    String[] expectedIdentifiers = { "LBR", "RBR", "COLON", "NEW\_LINE" };

    iterateAndVerify(tokenIterator, expectedIdentifiers);

    token = tokenIterator.next();

    while (token.getType().matches("(TAB)|(SPACE)")) {

      if (token.getType().equals("TAB")) {

        spacesAndTabs[1]++;

      } else {

        spacesAndTabs[0]++;

      }

      token = tokenIterator.next();

    }

    ASTNode statement = parseStatement(token, tokenIterator, spacesAndTabs);

    ASTNode declaration = new ASTNode(new Token(declarationName, "DECLARATION\_NAME", prev.getRow(), prev.getColumn()));

    declaration.appendChild(statement);

    statement.setParent(declaration);

    return declaration;

  }

  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 parseStatement(Token prev, Iterator<Token> tokenIterator, int[] spaceTabCount) throws ParseException {

    String[] numTypes = { "INT", "FLOAT", "HEX", "OCT", "BIN", "STRING" };

    if (spaceTabCount[0] + spaceTabCount[1] == 0) {

      throwAndPrint("Tab count is invalid", prev);

    }

    if (!prev.getType().equals("RETURN")) {

      throwAndPrint("Type is invalid", prev);

    }

    Token token = tokenIterator.next();

    if (!Arrays.asList(numTypes).contains(token.getType())) {

      throwAndPrint("Casting error", token);

    }

    ASTNode exp = parseExpression(token);

    ASTNode statement = new ASTNode(new Token("return", "RETURN", prev.getRow(), prev.getColumn()));

    statement.appendChild(exp);

    assert exp != null;

    exp.setParent(statement);

    token = tokenIterator.next();

    if (!token.getType().equals("NEW\_LINE")) {

      throwAndPrint("Type is invalid", token);

    }

    return statement;

  }

  private ASTNode parseIdentifier(Token prev, Iterator<Token> tokenIterator) throws ParseException {

    Token token = tokenIterator.next();

    if (token.getType().equals("LBR")) {

      return parseDefinitionCall(prev, tokenIterator);

    }

    throwAndPrint("Unexpected token", token);

    return null;

  }

  private ASTNode parseDefinitionCall(Token prev, Iterator<Token> tokenIterator) throws ParseException {

    String[] expectedIdentifiers = { "RBR", "NEW\_LINE" };

    iterateAndVerify(tokenIterator, expectedIdentifiers);

    ASTNode defCall = new ASTNode(new Token(prev.getValue(), "CALL", prev.getRow(), prev.getColumn()));

    if (!astMap.containsKey(prev.getValue())) {

      throwAndPrint("Unexpected token", prev);

    } else {

      defCall.appendChild(astMap.get(prev.getValue()).getRoot());

    }

    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);

  }

}

**Контрольний приклад**

def main():

    return 0b10111

def second(): # This line is going to be removed

    return 24.5

main PROC

mov ebx, 23

ret

main ENDP

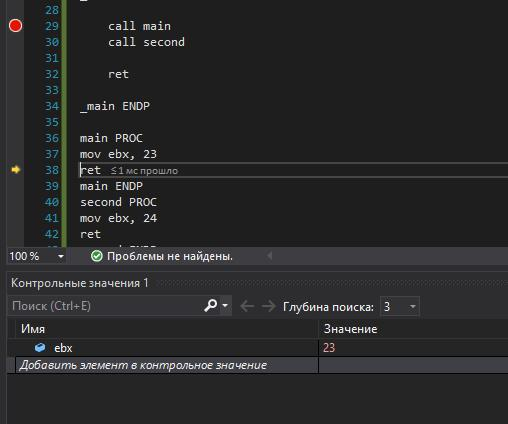
second PROC

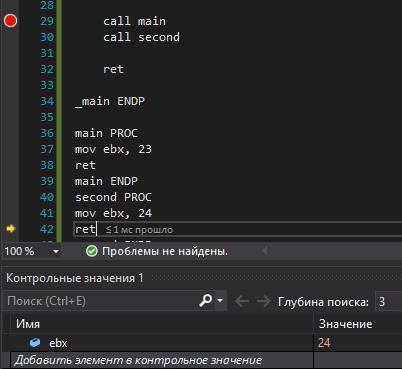
mov ebx, 24

ret

second ENDP

**Перевiрка роботи асемблерного коду:**

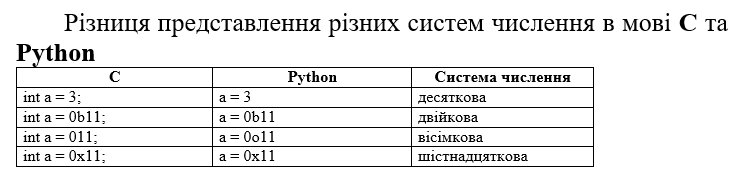




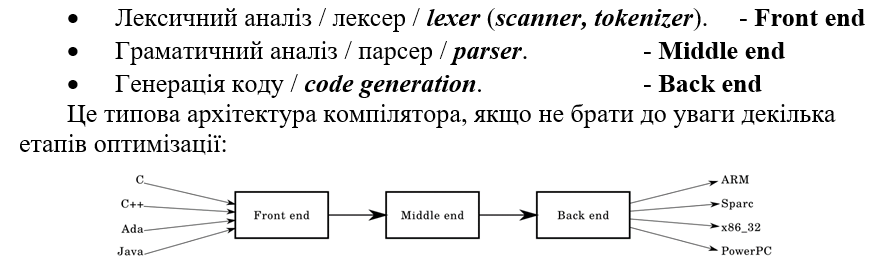
**Контрольні питання**

1. В чому суть і Incremental Approach? 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. І якщо необхідно щось кардинально змінити, це робиться одразу і без зайвих переписувань.
2. На які відмінності мови C та Python необхідно звернути увагу при виконанні роботи?

В мові Python кількість пробілів виступає у ролі ідентифікатора блоків на кшталт фігурних дужок {} в С. Стандартна кількість пробілів для виділення блоку (тіла функції в даному випадку) – чотири, але фактично їх може бути скільки завгодно. Програма на мові С може бути написана в один рядок. У випадку Python для написання декількох інструкцій в одному рядку необхідно розділити їх крапкою з комою «;».



1. Чому не слід використовувати складні регулярні вирази при написанні компілятора? Цей спосіб не є універсальним. (Також якщо парсити весь код регулярками це може вийти навіть більше затратним аніж лексичний перебір символів за правилами мови)
2. На які етапи поділяється розробка компілятора?



1. Що таке токен? Найменша одиниця мови, яку може зрозуміти парсер
2. В чому різниця між граматикою та AST?

**Граматика** визначає, як набір токенів буде пов’язаний між собою для формування мовної конструкції та як перетворити набір токенів у синтаксичне дерево. Граматика визначає структуру мови, описує її за допомогою правил.

**AST** – це інший спосіб представлення структури програми, яке має наступні відмінності від дерева розбору:

* видалення більшості нетермінальних вузлів, що мають одного нащадка;
* заміна частини термінальних символів атрибутами вузлів;
* модифікація групуючих вузлів.