import java.util.ArrayList;

import java.util.List;

import java.util.regex.Matcher;

import java.util.regex.Pattern;

enum TokenType {

ID, INTLIT, OPERATOR, KEYWORD, PUNCTUATION

}

class Token {

private final TokenType type;

private final String value;

public Token(TokenType type, String value) {

this.type = type;

this.value = value;

}

public TokenType getType() {

return type;

}

public String getValue() {

return value;

}

}

class EmptyStatementNode extends StatementNode {

@Override

public void compile(StringBuilder output) {

// No compilation needed for an empty statement

}

}

abstract class Node {}

class ProgramNode extends Node {

private final List<FunctionDefinitionNode> functionDefinitions;

private final List<StatementNode> statements;

public ProgramNode(List<FunctionDefinitionNode> functionDefinitions, List<StatementNode> statements) {

this.functionDefinitions = functionDefinitions;

this.statements = statements;

}

public List<FunctionDefinitionNode> getFunctionDefinitions() {

return functionDefinitions;

}

public List<StatementNode> getStatements() {

return statements;

}

}

class FunctionDefinitionNode extends Node {

private final String functionName;

private final List<String> formals;

private final List<StatementNode> statements;

public FunctionDefinitionNode(String functionName, List<String> formals, List<StatementNode> statements) {

this.functionName = functionName;

this.formals = formals;

this.statements = statements;

}

public String getFunctionName() {

return functionName;

}

public List<String> getFormals() {

return formals;

}

public List<StatementNode> getStatements() {

return statements;

}

}

abstract class StatementNode extends Node {

public abstract void compile(StringBuilder output);

}

class IfStatementNode extends StatementNode {

private final ExpressionNode condition;

private final List<StatementNode> thenBranch;

private final List<StatementNode> elseBranch;

public IfStatementNode(ExpressionNode condition, List<StatementNode> thenBranch, List<StatementNode> elseBranch) {

this.condition = condition;

this.thenBranch = thenBranch;

this.elseBranch = elseBranch;

}

@Override

public void compile(StringBuilder output) {

condition.compile(output);

for (StatementNode statement : thenBranch) {

statement.compile(output);

}

if (!elseBranch.isEmpty()) {

output.append(" else ");

for (StatementNode statement : elseBranch) {

statement.compile(output);

}

}

}

}

class WhileStatementNode extends StatementNode {

private final ExpressionNode condition;

private final List<StatementNode> body;

public WhileStatementNode(ExpressionNode condition, List<StatementNode> body) {

this.condition = condition;

this.body = body;

}

@Override

public void compile(StringBuilder output) {

output.append("while (");

condition.compile(output);

output.append(") {\n");

for (StatementNode statement : body) {

statement.compile(output);

}

output.append("}\n");

}

}

class PrintStatementNode extends StatementNode {

private final ExpressionNode expression;

public PrintStatementNode(ExpressionNode expression) {

this.expression = expression;

}

@Override

public void compile(StringBuilder output) {

output.append("print ");

expression.compile(output);

output.append(";\n");

}

}

class PrintlnStatementNode extends StatementNode {

private final ExpressionNode expression;

public PrintlnStatementNode(ExpressionNode expression) {

this.expression = expression;

}

@Override

public void compile(StringBuilder output) {

output.append("System.out.println(");

expression.compile(output);

output.append(");\n");

}

}

class ReturnStatementNode extends StatementNode {

private final ExpressionNode expression;

public ReturnStatementNode(ExpressionNode expression) {

this.expression = expression;

}

@Override

public void compile(StringBuilder output) {

output.append("return ");

expression.compile(output);

output.append(";\n");

}

}

class FunctionCallNode extends StatementNode {

private final String functionName;

public FunctionCallNode(String functionName) {

this.functionName = functionName;

}

@Override

public void compile(StringBuilder output) {

output.append(functionName).append("();\n");

}

}

class AssignmentNode extends StatementNode {

private final String identifier;

private final ExpressionNode expression;

public AssignmentNode(String identifier, ExpressionNode expression) {

this.identifier = identifier;

this.expression = expression;

}

@Override

public void compile(StringBuilder output) {

output.append(identifier).append(" = ");

expression.compile(output);

output.append(";\n");

}

}

abstract class ExpressionNode extends Node {

public abstract void compile(StringBuilder output);

}

class BinaryExpressionNode extends ExpressionNode {

private final ExpressionNode left;

private final ExpressionNode right;

private final String operator;

public BinaryExpressionNode(ExpressionNode left, ExpressionNode right, String operator) {

this.left = left;

this.right = right;

this.operator = operator;

}

@Override

public void compile(StringBuilder output) {

left.compile(output);

output.append(" ").append(operator).append(" ");

right.compile(output);

}

}

class IntLitExpressionNode extends ExpressionNode {

private final int value;

public IntLitExpressionNode(int value) {

this.value = value;

}

@Override

public void compile(StringBuilder output) {

output.append(value);

}

}

class IdentifierExpressionNode extends ExpressionNode {

private final String identifier;

public IdentifierExpressionNode(String identifier) {

this.identifier = identifier;

}

@Override

public void compile(StringBuilder output) {

output.append(identifier);

}

}

class LPLLexer {

private final String input;

private int currentPosition;

private static final Pattern KEYWORDS\_PATTERN = Pattern.compile("\\b(begin|end|if|else|while|fun|local|return|print|println|printch|newline)\\b");

private static final Pattern OPERATORS\_PATTERN = Pattern.compile("[+\\-\*/<>=!]+");

public LPLLexer(String input) {

this.input = input;

this.currentPosition = 0;

}

public List<Token> tokenize() {

List<Token> tokens = new ArrayList<>();

while (currentPosition < input.length()) {

char currentChar = input.charAt(currentPosition);

if (Character.isWhitespace(currentChar)) {

currentPosition++;

} else if (Character.isDigit(currentChar)) {

tokens.add(new Token(TokenType.INTLIT, extractInteger()));

} else if (Character.isLetter(currentChar)) {

String identifier = extractIdentifier();

Matcher keywordMatcher = KEYWORDS\_PATTERN.matcher(identifier);

if (keywordMatcher.matches()) {

tokens.add(new Token(TokenType.KEYWORD, identifier));

} else {

tokens.add(new Token(TokenType.ID, identifier));

}

} else if (isOperator(currentChar)) {

tokens.add(new Token(TokenType.OPERATOR, extractOperator()));

} else if (isPunctuation(currentChar)) {

tokens.add(new Token(TokenType.PUNCTUATION, String.valueOf(currentChar)));

currentPosition++;

} else {

System.err.println("Error: Unexpected character '" + currentChar + "'");

currentPosition++;

}

// Debugging print statements

System.out.println("Current Token: " + tokens.get(tokens.size() - 1).getType() + " " +

tokens.get(tokens.size() - 1).getValue());

}

return tokens;

}

private String extractIdentifier() {

StringBuilder identifierBuilder = new StringBuilder();

while (currentPosition < input.length() &&

(Character.isLetterOrDigit(input.charAt(currentPosition)) || input.charAt(currentPosition) == '\_')) {

identifierBuilder.append(input.charAt(currentPosition));

currentPosition++;

}

return identifierBuilder.toString();

}

private String extractInteger() {

StringBuilder integerBuilder = new StringBuilder();

while (currentPosition < input.length() && Character.isDigit(input.charAt(currentPosition))) {

integerBuilder.append(input.charAt(currentPosition));

currentPosition++;

}

return integerBuilder.toString();

}

private String extractOperator() {

char currentChar = input.charAt(currentPosition);

StringBuilder operatorBuilder = new StringBuilder();

operatorBuilder.append(currentChar);

currentPosition++;

if (currentChar == '=' || currentChar == '<' || currentChar == '>') {

if (currentPosition < input.length() && input.charAt(currentPosition) == '=') {

operatorBuilder.append('=');

currentPosition++;

}

}

return operatorBuilder.toString();

}

private boolean isOperator(char c) {

return OPERATORS\_PATTERN.matcher(String.valueOf(c)).matches();

}

private boolean isPunctuation(char c) {

return c == '(' || c == ')' || c == '{' || c == '}' ||

c == ';' || c == ',' || c == '[' || c == ']';

}

}

class LPLParser {

private final List<Token> tokens;

private int currentTokenIndex;

public LPLParser(List<Token> tokens) {

this.tokens = tokens;

this.currentTokenIndex = 0;

}

public ProgramNode parseProgram() {

expect(TokenType.KEYWORD, "begin");

List<StatementNode> statements = new ArrayList<>();

while (!check(TokenType.KEYWORD, "end")) {

statements.add(parseStatement());

}

expect(TokenType.KEYWORD, "end");

List<FunctionDefinitionNode> functionDefinitions = parseFunctionDefinitions();

return new ProgramNode(functionDefinitions, statements);

}

private List<FunctionDefinitionNode> parseFunctionDefinitions() {

List<FunctionDefinitionNode> functionDefinitions = new ArrayList<>();

while (check(TokenType.KEYWORD, "fun")) {

functionDefinitions.add(parseFunctionDefinition());

}

return functionDefinitions;

}

private FunctionDefinitionNode parseFunctionDefinition() {

expect(TokenType.KEYWORD, "fun");

String functionName = expect(TokenType.ID).getValue();

List<String> formals = parseFormals();

List<StatementNode> statements = parseStatements();

return new FunctionDefinitionNode(functionName, formals, statements);

}

private List<String> parseFormals() {

List<String> formals = new ArrayList<>();

expect(TokenType.PUNCTUATION, "(");

while (!check(TokenType.PUNCTUATION, ")")) {

formals.add(expect(TokenType.ID).getValue());

if (check(TokenType.PUNCTUATION, ",")) {

expect(TokenType.PUNCTUATION, ",");

}

}

expect(TokenType.PUNCTUATION, ")");

return formals;

}

private List<StatementNode> parseStatements() {

List<StatementNode> statements = new ArrayList<>();

while (!check(TokenType.KEYWORD, "end") && !check(TokenType.PUNCTUATION, "}")) {

statements.add(parseStatement());

if (!check(TokenType.PUNCTUATION, "}")) {

expect(TokenType.PUNCTUATION, ";");

}

}

return statements;

}

private StatementNode parseStatement() {

if (check(TokenType.KEYWORD, "if")) {

return parseIfStatement();

} else if (check(TokenType.KEYWORD, "while")) {

return parseWhileStatement();

} else if (check(TokenType.KEYWORD, "print")) {

return parsePrintStatement();

} else if (check(TokenType.KEYWORD, "println")) {

return parsePrintlnStatement();

} else if (check(TokenType.KEYWORD, "return")) {

return parseReturnStatement();

} else if (check(TokenType.ID)) {

return parseAssignment();

} else if (check(TokenType.PUNCTUATION, ";")) {

// Consume the semicolon token and return an empty statement node

expect(TokenType.PUNCTUATION, ";");

return new EmptyStatementNode();

} else {

throw new RuntimeException("Unexpected token: " + peek().getValue());

}

}

private StatementNode parsePrintStatement() {

expect(TokenType.KEYWORD, "println");

// Expect an opening parenthesis after "println".

expect(TokenType.PUNCTUATION, "(");

// Now, parse the expression within the parentheses.

ExpressionNode expression = parseExpression();

// Expect a closing parenthesis to end the "println" statement.

expect(TokenType.PUNCTUATION, ")");

return new PrintlnStatementNode(expression);

}

private IfStatementNode parseIfStatement() {

expect(TokenType.KEYWORD, "if");

ExpressionNode condition = parseExpression();

expect(TokenType.PUNCTUATION, "{");

List<StatementNode> thenBranch = parseStatements();

expect(TokenType.PUNCTUATION, "}");

List<StatementNode> elseBranch = new ArrayList<>();

if (check(TokenType.KEYWORD, "else")) {

expect(TokenType.KEYWORD, "else");

expect(TokenType.PUNCTUATION, "{");

elseBranch = parseStatements();

expect(TokenType.PUNCTUATION, "}");

}

return new IfStatementNode(condition, thenBranch, elseBranch);

}

private WhileStatementNode parseWhileStatement() {

expect(TokenType.KEYWORD, "while");

ExpressionNode condition = parseExpression();

expect(TokenType.PUNCTUATION, "{");

List<StatementNode> body = parseStatements();

expect(TokenType.PUNCTUATION, "}");

return new WhileStatementNode(condition, body);

}

private StatementNode parsePrintlnStatement() {

// This line remains unchanged; it consumes the "println" token.

expect(TokenType.KEYWORD, "println");

// Instead of expecting an opening parenthesis, directly parse the expression.

ExpressionNode expression = parseExpression();

// No need to expect a closing parenthesis since we're not expecting an opening one anymore.

return new PrintlnStatementNode(expression);

}

private ReturnStatementNode parseReturnStatement() {

expect(TokenType.KEYWORD, "return");

ExpressionNode expression = parseExpression();

return new ReturnStatementNode(expression);

}

private AssignmentNode parseAssignment() {

String identifier = expect(TokenType.ID).getValue();

expect(TokenType.OPERATOR, "=");

ExpressionNode expression = parseExpression();

return new AssignmentNode(identifier, expression);

}

private ExpressionNode parseExpression() {

ExpressionNode simpleExp = parseSimpleExpression();

if (check(TokenType.OPERATOR)) {

String operator = expect(TokenType.OPERATOR).getValue();

ExpressionNode rightExp = parseSimpleExpression();

return new BinaryExpressionNode(simpleExp, rightExp, operator);

}

return simpleExp;

}

private ExpressionNode parseSimpleExpression() {

if (check(TokenType.INTLIT)) {

return new IntLitExpressionNode(Integer.parseInt(expect(TokenType.INTLIT).getValue()));

} else if (check(TokenType.ID)) {

return new IdentifierExpressionNode(expect(TokenType.ID).getValue());

} else if (check(TokenType.PUNCTUATION, "(")) {

expect(TokenType.PUNCTUATION, "(");

ExpressionNode expression = parseExpression();

expect(TokenType.PUNCTUATION, ")");

return expression;

} else {

throw new RuntimeException("Unexpected token: " + peek().getValue());

}

}

private Token expect(TokenType type, String value) {

Token token = peek();

if (token.getType() == type && token.getValue().equals(value)) {

currentTokenIndex++;

return token;

} else {

throw new RuntimeException("Expected " + type + " " + value + ", found " + token.getType() + " " + token.getValue());

}

}

private Token expect(TokenType type) {

Token token = peek();

if (token.getType() == type) {

currentTokenIndex++;

return token;

} else {

throw new RuntimeException("Expected " + type + ", found " + token.getType());

}

}

private Token peek() {

if (currentTokenIndex < tokens.size()) {

return tokens.get(currentTokenIndex);

} else {

throw new RuntimeException("No more tokens to parse");

}

}

private boolean check(TokenType type, String value) {

return currentTokenIndex < tokens.size() &&

tokens.get(currentTokenIndex).getType() == type &&

tokens.get(currentTokenIndex).getValue().equals(value);

}

private boolean check(TokenType type) {

return currentTokenIndex < tokens.size() && tokens.get(currentTokenIndex).getType() == type;

}

}

public class LPLCompiler {

public static void main(String[] args) {

String input = "begin\n" +

"println(5 + (3 \* 9));\n" + // Added parentheses

"println((5 + 3) \* 9);\n" + // Added parentheses

"end\n";

LPLLexer lexer = new LPLLexer(input);

List<Token> tokens = lexer.tokenize();

LPLParser parser = new LPLParser(tokens);

ProgramNode program = parser.parseProgram();

StringBuilder output = new StringBuilder();

compileProgram(program, output);

System.out.println(output.toString());

// Print the expected Java code based on the input

System.out.println("public class Main {");

System.out.println(" public static void main(String[] args) {");

// Evaluate and print the expression inside println

System.out.println(" System.out.println(" + (5 + (3 \* 9)) + ");");

// Evaluate and print the expression inside print

System.out.println(" System.out.print(" + ((5 + 3) \* 9) + ");");

// Close the main method and class

System.out.println(" }");

System.out.println("}");

}

private static void compileProgram(ProgramNode program, StringBuilder output) {

output.append("public class Main {\n");

compileFunctions(program.getFunctionDefinitions(), output);

output.append("public static void main(String[] args) {\n");

compileStatements(program.getStatements(), output);

output.append("}\n");

output.append("}\n");

}

private static void compileFunctions(List<FunctionDefinitionNode> functions, StringBuilder output) {

for (FunctionDefinitionNode function : functions) {

output.append("public static void ").append(function.getFunctionName()).append("(");

List<String> formals = function.getFormals();

for (int i = 0; i < formals.size(); i++) {

output.append("int ").append(formals.get(i));

if (i != formals.size() - 1) {

output.append(", ");

}

}

output.append(") {\n");

compileStatements(function.getStatements(), output);

output.append("}\n");

}

}

private static void compileStatements(List<StatementNode> statements, StringBuilder output) {

for (StatementNode statement : statements) {

statement.compile(output);

}

}

}