**CTE711S COMPILER TECHNIQUES, SEM 1 2025**

***Group/Team should edit this document to suit their Group Assignment project topic***

***The format of this MS Word file should be: Font: Calibri, Font size 12, line spacing: single***

**GROUP ASSIGNMENT TEAM LIST AND PROJECT SUBMISSION TEMPLATE**

*The Assignment must be done as a group, but each student/group member must submit the MS Word file on elearning and also send the file to the email:*[*postgraduatementor@gmail.com*](mailto:postgraduatementor@gmail.com) *on or before 23h59 on Friday, 2 May 2025.*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **SN** | **NAME** | **STUDENT NUMBER** | **SPECIALISATION** | **Programme** |
| 1 | Mario Uushunga **(TL)** | 223023795 | Computer Science (Software Development) | Full-Time |
| 2 | Prince Itope | 223102792 | Computer Science (Software Development) | Full-Time |
| 3 | Diago Flavio De Oliveira | 223086525 | Computer Science (Software Development) | Full-Time |
| 4 | Benjamin Nehoya | 223056553 | Computer Science (Software Development) | Full-Time |
| 5 | Gospel Nwagbara | 223116866 | Computer Science (Software Development) | Full-Time |

**SUBMITTED BY :** Mario Uushunga

**TITLE OF PROJECT:**

Development of a Mini Compiler System Using Java

**DATE**:

02 May 2025

**THE JAVA PROGRAM SOURCE CODES**

import java.util.\*;  
  
public class CompilerMain {  
 private static final Set<String> *KEYWORDS* =  
 Set.*of*("BEGIN", "INTEGER", "LET", "INPUT", "WRITE", "END");  
  
 public static void main(String[] args) {  
 Scanner scanner = new Scanner(System.*in*);  
  
 Stage1Lexer lexer = new Stage1Lexer();  
 Stage2Parser parser = new Stage2Parser();  
 Stage3SemanticAnalyzer semantic = new Stage3SemanticAnalyzer();  
 Stage4ICRGenerator icrGen = new Stage4ICRGenerator();  
 Stage5CodeGenerator codeGen = new Stage5CodeGenerator();  
 Stage6Optimizer optimizer = new Stage6Optimizer();  
 Stage7TMCGenerator tmcGen = new Stage7TMCGenerator();  
  
 String mode;  
 while (true) {  
 System.*out*.println("Select mode: 1) Iterative (line-by-line) 2) Batch (all at once)");  
 mode = scanner.nextLine().trim();  
 if (mode.equals("1") || mode.equals("2")) break;  
 }  
  
 if (mode.equals("1")) {  
 int lineNum = 1;  
 while (true) {  
 System.*out*.print("Enter Line " + lineNum + " (or 99 to quit): ");  
 String input = scanner.nextLine().trim();  
 if (input.equals("99")) break;  
 System.*out*.println("Line " + lineNum++ + ": " + input);  
  
 List<String> tokens = lexer.analyze(input);  
  
 if (!tokens.isEmpty()  
 && *KEYWORDS*.contains(tokens.get(0))  
 && !tokens.contains("=")) {  
 tmcGen.generateRaw(tokens);  
 System.*out*.println("======END OF COMPILATION======\n");  
 continue;  
 }  
  
 int eq = tokens.indexOf("=");  
 List<String> exprTokens;  
 String expr;  
 if (eq != -1) {  
 exprTokens = tokens.subList(eq + 1, tokens.size());  
 expr = input.substring(input.indexOf('=') + 1).trim();  
 } else {  
  
 exprTokens = tokens;  
 expr = input;  
 }  
  
 Stage2Parser.DerivationType dt = *chooseDerivation*(scanner);  
 if (!parser.parse(exprTokens, dt)) continue;  
 if (!semantic.analyze(exprTokens)) continue;  
  
 List<String> icr = icrGen.generate(expr);  
 codeGen.generate(icr);  
 List<String> opt = optimizer.optimize(icr);  
 tmcGen.generate(opt);  
  
 System.*out*.println("======END OF COMPILATION======\n");  
 }  
  
 } else {  
 List<String> inputs = new ArrayList<>();  
 int lineNum = 1;  
 while (true) {  
 System.*out*.print("Enter Line " + lineNum + " (or 99 to finish): ");  
 String input = scanner.nextLine().trim();  
 if (input.equals("99")) break;  
 inputs.add(input);  
 lineNum++;  
 }  
  
 lineNum = 1;  
 for (String input : inputs) {  
 System.*out*.println("Line " + lineNum++ + ": " + input);  
  
 List<String> tokens = lexer.analyze(input);  
  
 if (!tokens.isEmpty()  
 && *KEYWORDS*.contains(tokens.get(0))  
 && !tokens.contains("=")) {  
 tmcGen.generateRaw(tokens);  
 System.*out*.println("======END OF COMPILATION======\n");  
 continue;  
 }  
  
 int eq = tokens.indexOf("=");  
 List<String> exprTokens;  
 String expr;  
 if (eq != -1) {  
 exprTokens = tokens.subList(eq + 1, tokens.size());  
 expr = input.substring(input.indexOf('=') + 1).trim();  
 } else {  
 exprTokens = tokens;  
 expr = input;  
 }  
  
 Stage2Parser.DerivationType dt = *chooseDerivation*(scanner);  
 if (!parser.parse(exprTokens, dt)) continue;  
 if (!semantic.analyze(exprTokens)) continue;  
  
 List<String> icr = icrGen.generate(expr);  
 codeGen.generate(icr);  
 List<String> opt = optimizer.optimize(icr);  
 tmcGen.generate(opt);  
  
 System.*out*.println("======END OF COMPILATION======\n");  
 }  
 }  
  
 scanner.close();  
 System.*out*.println("Exiting Compiler...");  
 }  
  
 private static Stage2Parser.DerivationType chooseDerivation(Scanner scanner) {  
 while (true) {  
 System.*out*.println("Choose derivation type: 1) Leftmost (top-down) 2) Rightmost (bottom-up)");  
 String choice = scanner.nextLine().trim();  
 if (choice.equals("1")) return Stage2Parser.DerivationType.*LEFTMOST*;  
 if (choice.equals("2")) return Stage2Parser.DerivationType.*RIGHTMOST*;  
 }  
 }  
}

import java.util.\*;  
  
public class Stage1Lexer {  
 public List<String> analyze(String input) {  
 System.*out*.println("======STAGE1: COMPILER TECHNIQUES--> LEXICAL ANALYSIS-Scanner");  
 System.*out*.println("SYMBOL TABLE COMPRISING ATTRIBUTES AND TOKENS:\n");  
 Set<String> KEYWORDS = Set.*of*("BEGIN", "INTEGER", "LET", "INPUT", "WRITE", "END");  
 Set<String> OPERATORS = Set.*of*("+", "-", "\*", "/");  
 StringTokenizer st = new StringTokenizer(input, "+-\*/=;, ", true);  
 List<String> tokens = new ArrayList<>();  
 String prev = null;  
 while (st.hasMoreTokens()) {  
 String t = st.nextToken();  
 if (t.trim().isEmpty() || t.equals(",")) continue;  
 if (prev != null && OPERATORS.contains(prev) && OPERATORS.contains(t)) {  
 System.out.printf("SYNTAX ERROR – combined operators '%s%s' not allowed%n", prev, t);  
 }  
 tokens.add(t);  
 prev = t;  
 }  
 if (input.trim().endsWith(";")) {  
 System.out.println("SYNTAX ERROR – semicolon ';' at end of line not allowed");  
 }  
 for (int i = 0; i < tokens.size(); i++) {  
 String tok = tokens.get(i);  
 String type;  
 if (tok.matches("\\d")) {  
 System.out.printf("SYNTAX ERROR – numbers not allowed: '%s'%n", tok);  
 type = "Invalid";  
 } else if (OPERATORS.contains(tok)) {  
 type = "Operator";  
 } else if ("=".equals(tok) || ";".equals(tok)) {  
 type = "Symbol";  
 } else if (tok.matches("[A-Z]{2,}")) {  
 if (KEYWORDS.contains(tok)) type = "Keyword";  
 else {  
 System.out.printf("LEXICAL ERROR – keyword '%s' not recognized%n", tok);  
 type = "Invalid";  
 }  
 } else if (tok.matches("[A-Za-z]") || tok.matches("[a-z]+")) {  
 type = "Identifier";  
 } else {  
 System.out.printf("SYNTAX ERROR – invalid character '%s'%n", tok);  
 type = "Invalid";  
 }  
 System.out.printf("TOKEN#%d %-7s %s%n", i + 1, tok, type);  
 }  
 System.out.println();  
 return tokens;  
 }  
}

import java.util.\*;  
  
public class Stage2Parser {  
 public enum DerivationType {*LEFTMOST*, *RIGHTMOST*}  
  
 private static class DerivationStep {  
 String line, rule;  
  
 DerivationStep(String l, String r) {  
 line = l;  
 rule = r;  
 }  
 }  
  
 public boolean parse(List<String> tokens, DerivationType type) {  
 System.*out*.println("======STAGE2: COMPILER TECHNIQUES--> SYNTAX ANALYSIS-Parser");  
 String input = String.*join*("", tokens);  
 System.*out*.println("GET THE DERIVATION FOR : " + input + "\n");  
  
 List<String> errors = new ArrayList<>();  
 int e = 0;  
  
 for (int i = 0; i + 1 < tokens.size(); i++) {  
 if ("+-\*/".contains(tokens.get(i)) && "+-\*/".contains(tokens.get(i + 1))) {  
 errors.add("#" + (++e) + " SYNTAX ERROR – combined operators '"  
 + tokens.get(i) + tokens.get(i + 1) + "' not allowed");  
 }  
 }  
  
 if (!tokens.isEmpty() && tokens.get(tokens.size() - 1).equals(";")) {  
 errors.add("#" + (++e) + " SYNTAX ERROR – semicolon ';' at end of line not allowed");  
 }  
  
 for (String t : tokens) {  
 if (t.matches("\\d+")) {  
 errors.add("#" + (++e) + " SYNTAX ERROR – numbers not allowed: '" + t + "'");  
 }  
 }  
  
 for (String t : tokens) {  
 if (!t.matches("[A-Za-z]") && !"+-\*/".contains(t)  
 && !"=".equals(t) && !";".equals(t)) {  
 errors.add("#" + (++e) + " SYNTAX ERROR – invalid token '" + t + "'");  
 }  
 }  
  
 for (int i = 0; i < tokens.size(); i++) {  
 String t = tokens.get(i);  
 if (t.length() == 1 && "%$&<>;".contains(t)) {  
 errors.add("#" + (++e) + " SEMANTIC ERROR – symbol '" + t + "' not allowed at position " + (i + 1));  
 }  
 }  
  
 if (!errors.isEmpty()) {  
 errors.forEach(System.*out*::println);  
 System.*out*.println();  
 return false;  
 }  
  
 List<String> ops = new ArrayList<>(), ids = new ArrayList<>();  
 for (String t : tokens) {  
 if ("+-\*/".contains(t)) ops.add(t);  
 else if (t.matches("[A-Za-z]")) ids.add(t);  
 }  
  
 List<DerivationStep> steps = new ArrayList<>();  
 steps.add(new DerivationStep("E -> E", "Rule1 (R1)"));  
 String seq = "E";  
 for (String op : ops) {  
 seq += op + "E";  
 steps.add(new DerivationStep("E -> " + seq, "Rule2 (R2)"));  
 }  
 for (int i = 0; i < ids.size(); i++) {  
 int idx = Character.*toUpperCase*(ids.get(i).charAt(0)) - 'A' + 1;  
 seq = replaceNth(seq, "E", "E" + idx, i);  
 steps.add(new DerivationStep("E -> " + seq, "Rule3 (R3)"));  
 }  
 for (String id : ids) {  
 int idx = Character.*toUpperCase*(id.charAt(0)) - 'A' + 1;  
 seq = seq.replaceFirst("E" + idx, id);  
 steps.add(new DerivationStep("E -> " + seq, "Rule4 (R4)"));  
 }  
 if (type == DerivationType.*RIGHTMOST*) Collections.*reverse*(steps);  
 steps.forEach(s -> System.*out*.printf("%-50s%s%n", s.line, s.rule));  
 System.*out*.println();  
 return true;  
 }  
  
 private int nthIndex(String s, String sub, int n) {  
 int pos = -1, from = 0;  
 for (int i = 0; i <= n; i++) {  
 pos = s.indexOf(sub, from);  
 if (pos < 0) return -1;  
 from = pos + sub.length();  
 }  
 return pos;  
 }  
  
 private String replaceNth(String s, String find, String rep, int n) {  
 int p = nthIndex(s, find, n);  
 if (p < 0) return s;  
 return s.substring(0, p) + rep + s.substring(p + find.length());  
 }  
}

import java.util.\*;  
  
public class Stage3SemanticAnalyzer {  
 public boolean analyze(List<String> tokens) {  
 System.*out*.println("======STAGE3: COMPILER TECHNIQUES--> SEMANTIC ANALYSIS");  
 Set<Character> FORBIDDEN = Set.*of*('%', '$', '&', '<', '>', ';');  
 int e = 0;  
 for (int i = 0; i < tokens.size(); i++) {  
 String t = tokens.get(i);  
 if (t.length() == 1 && FORBIDDEN.contains(t.charAt(0))) {  
 System.*out*.printf("#%d SEMANTIC ERROR – symbol '%s' not allowed at position %d%n", ++e, t, i + 1);  
 }  
 }  
 if (e > 0) {  
 System.*out*.println();  
 return false;  
 }  
 String expr = String.*join*("", tokens);  
 System.*out*.println("CONCLUSION --> This expression: " + expr + " is Syntactically and Semantically correct\n");  
 return true;  
 }  
}

import java.util.\*;  
  
public class Stage4ICRGenerator {  
 public List<String> generate(String expr) {  
 String clean = expr.endsWith(";") ? expr.substring(0, expr.length() - 1) : expr;  
 System.*out*.println("======STAGE4: INTERMEDIATE CODE REPRESENTATION (ICR)");  
 System.*out*.println("INPUT STRING: " + clean);  
 System.*out*.println("BODMAS\n");  
  
 StringTokenizer st = new StringTokenizer(clean, "+-\*/", true);  
 List<String> tokens = new ArrayList<>();  
 while (st.hasMoreTokens()) {  
 String t = st.nextToken().trim();  
 if (!t.isEmpty()) tokens.add(t);  
 }  
  
 List<String> icr = new ArrayList<>();  
 int tCount = 1;  
 boolean done;  
 do {  
 done = false;  
 for (int i = 0; i < tokens.size(); i++) {  
 String op = tokens.get(i);  
 if ("\*/".contains(op)) {  
 String l = tokens.get(i - 1), r = tokens.get(i + 1);  
 String tmp = "t" + tCount++;  
 icr.add(tmp + " = " + l + " " + op + " " + r);  
 tokens.set(i - 1, tmp);  
 tokens.remove(i + 1);  
 tokens.remove(i);  
 done = true;  
 break;  
 }  
 }  
 } while (done);  
  
 for (int i = 0; i < tokens.size(); i++) {  
 String op = tokens.get(i);  
 if ("+-".contains(op)) {  
 String l = tokens.get(i - 1), r = tokens.get(i + 1);  
 String tmp = "t" + tCount++;  
 icr.add(tmp + " = " + l + " " + op + " " + r);  
 tokens.set(i - 1, tmp);  
 tokens.remove(i + 1);  
 tokens.remove(i);  
 i = -1;  
 }  
 }  
  
 List<String> temps = new ArrayList<>();  
 for (int i = 1; i < tCount; i++) temps.add("t" + i);  
 System.out.println("TEMPORARY VARIABLES: WE USE: " +  
 String.join(", ", temps) + "\n");  
  
 icr.forEach(System.out::println);  
 System.out.println();  
 return icr;  
 }  
}

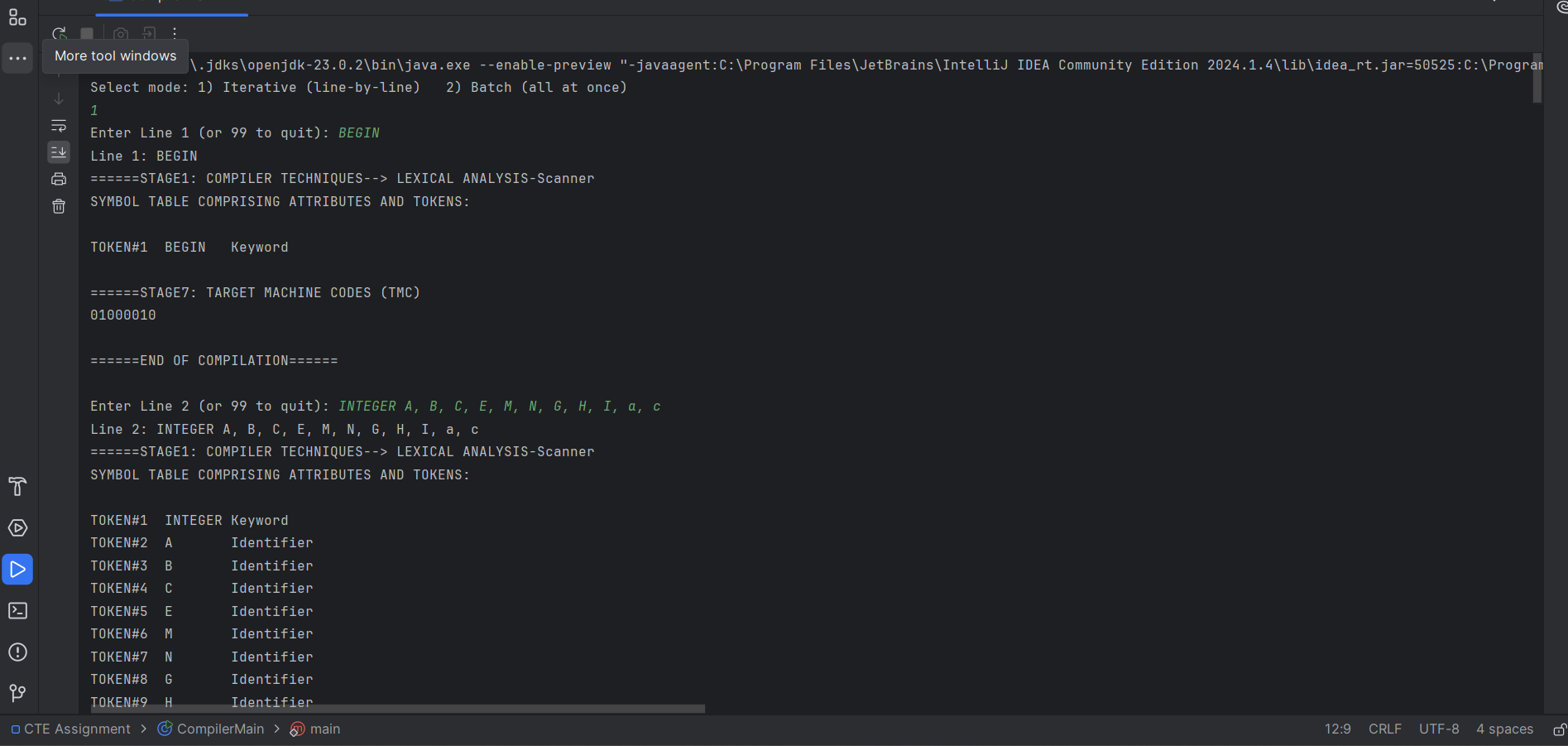
import java.util.\*;  
import java.util.regex.\*;  
  
public class Stage5CodeGenerator {  
 public List<String> generate(List<String> icr) {  
 System.*out*.println("======STAGE5: CODE GENERATION (CG)");  
 System.*out*.println("WE USE: LDA, MUL, ADD, SUB, STR\n");  
  
 List<String> code = new ArrayList<>();  
 Pattern p = Pattern.*compile*("(.+?) = (.+?) ([+\\-\*/]) (.+)");  
 for (String instr : icr) {  
 Matcher m = p.matcher(instr);  
 if (!m.matches()) continue;  
 String dest = m.group(1).trim();  
 String l = m.group(2).trim();  
 String op = m.group(3);  
 String r = m.group(4).trim();  
  
 code.add("LDA " + l);  
 switch (op) {  
 case "\*":  
 code.add("MUL " + r);  
 break;  
 case "/":  
 code.add("DIV " + r);  
 break;  
 case "+":  
 code.add("ADD " + r);  
 break;  
 default:  
 code.add("SUB " + r);  
 }  
 code.add("STR " + dest);  
 }  
  
 code.forEach(System.*out*::println);  
 System.*out*.println();  
 return code;  
 }  
}

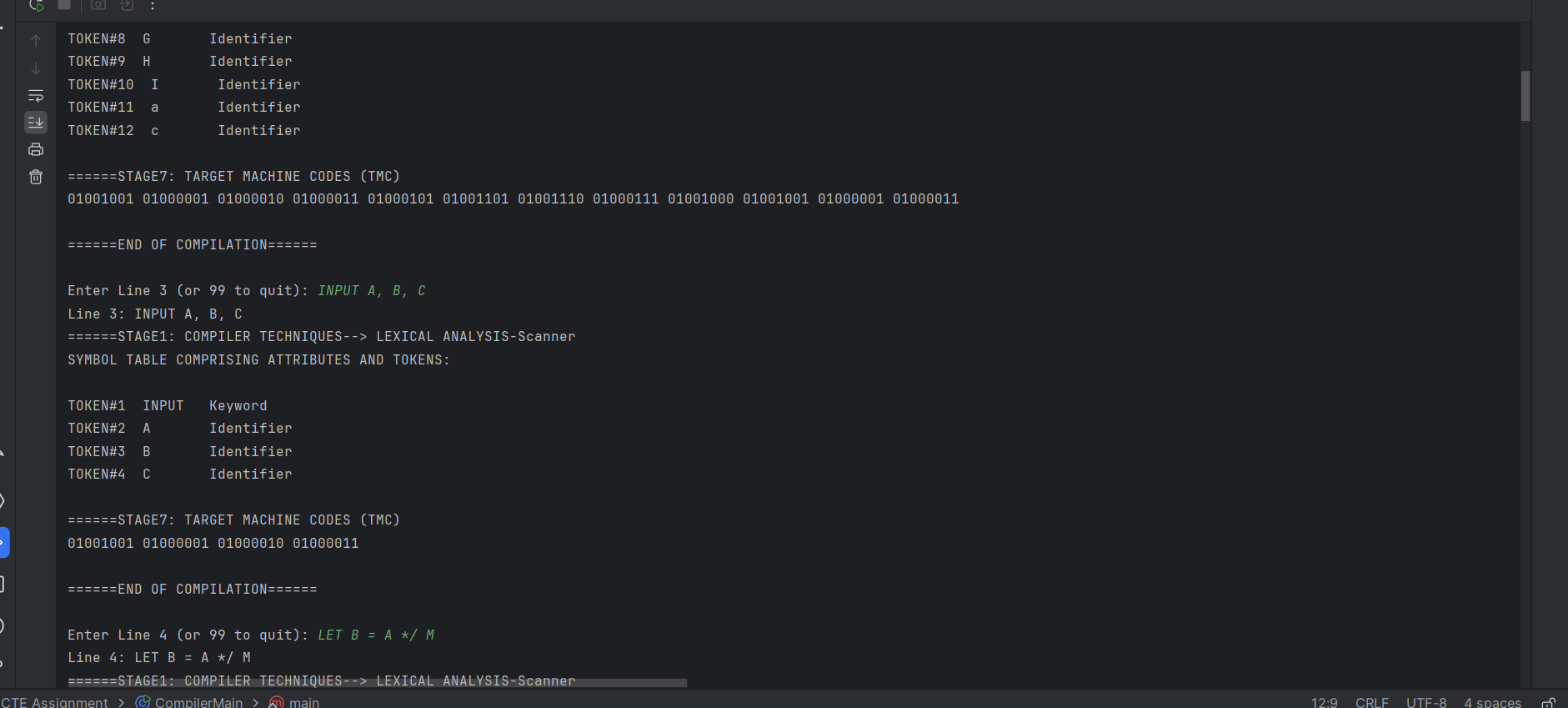
import java.util.\*;  
import java.util.regex.\*;  
  
public class Stage6Optimizer {  
 public List<String> optimize(List<String> icr) {  
 System.*out*.println("======STAGE6: CODE OPTIMISATION (CO)");  
 System.*out*.println("WE USE: LDA, MUL, ADD, SUB, STR, BODMAS\n");  
  
 List<String> opt = new ArrayList<>();  
 Pattern p = Pattern.*compile*("(t\\d+) = (.+?) ([+\\-\*/]) (.+)");  
 for (String instr : icr) {  
 Matcher m = p.matcher(instr);  
 if (!m.matches()) continue;  
 String dest = m.group(1);  
 String l = m.group(2);  
 String op = m.group(3);  
 String r = m.group(4);  
  
 String mn = switch (op) {  
 case "\*" -> "MUL";  
 case "/" -> "DIV";  
 case "+" -> "ADD";  
 default -> "SUB";  
 };  
 opt.add(String.*format*("%s %s, %s, %s", mn, dest, r, l));  
 }  
  
 opt.forEach(System.*out*::println);  
 System.*out*.println();  
 return opt;  
 }  
}

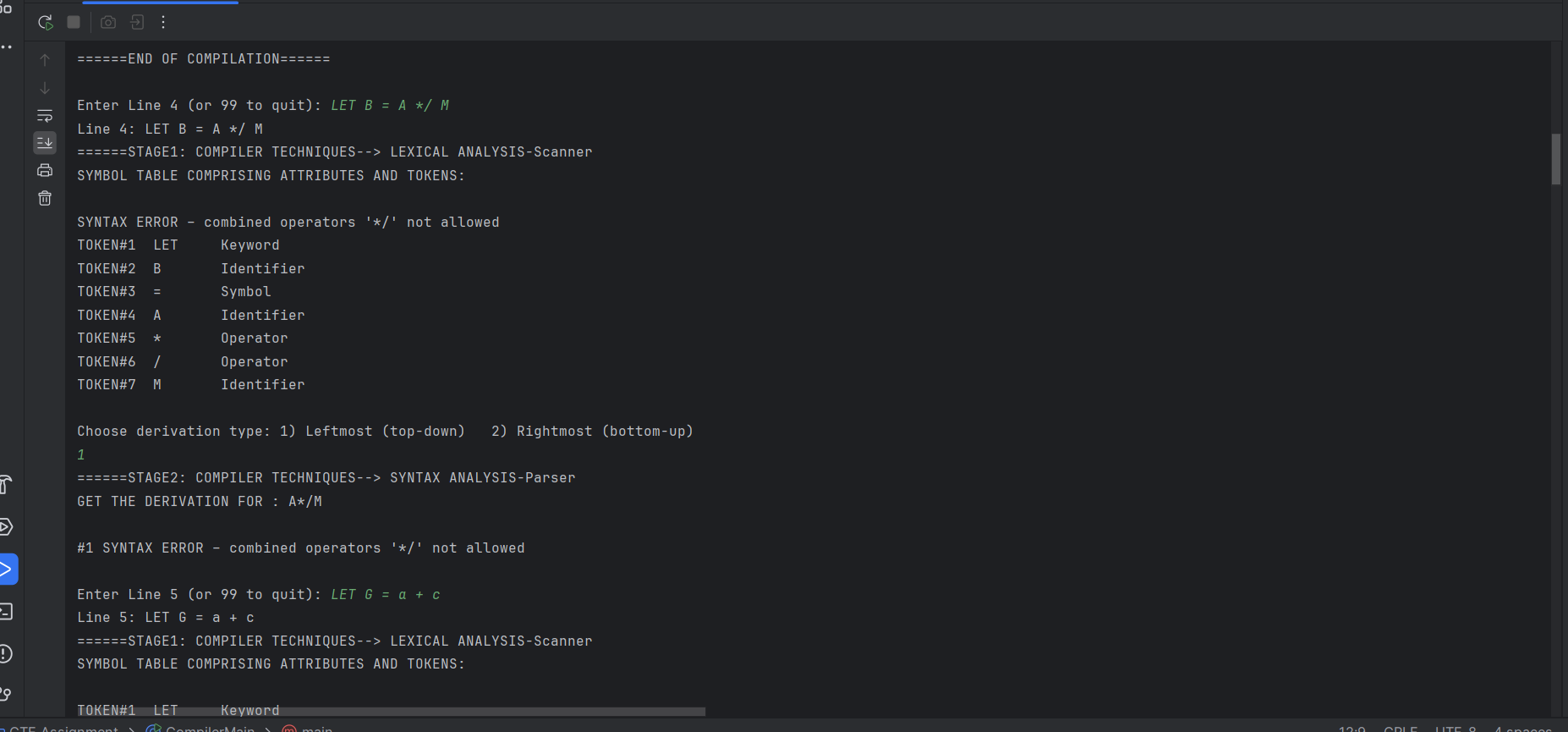
import java.util.\*;  
  
public class Stage7TMCGenerator {  
 public void generate(List<String> opt) {  
 System.*out*.println("======STAGE7: TARGET MACHINE CODES (TMC)");  
 for (String instr : opt) {  
 String[] tok = instr.split("[ ,]+");  
 List<String> binaries = new ArrayList<>();  
 for (String t : tok) {  
 char c = Character.*toUpperCase*(t.charAt(0));  
 binaries.add(to8(c));  
 }  
 System.*out*.println(String.*join*(" ", binaries));  
 }  
 System.*out*.println();  
 }  
  
 public void generateRaw(List<String> tokens) {  
 System.*out*.println("======STAGE7: TARGET MACHINE CODES (TMC)");  
 List<String> binaries = new ArrayList<>();  
 for (String t : tokens) {  
 char c = Character.*toUpperCase*(t.charAt(0));  
 binaries.add(to8(c));  
 }  
 System.*out*.println(String.*join*(" ", binaries));  
 System.*out*.println();  
 }  
  
 private String to8(char c) {  
 return String.*format*("%8s", Integer.*toBinaryString*(c & 0xFF))  
 .replace(' ', '0');  
 }  
}

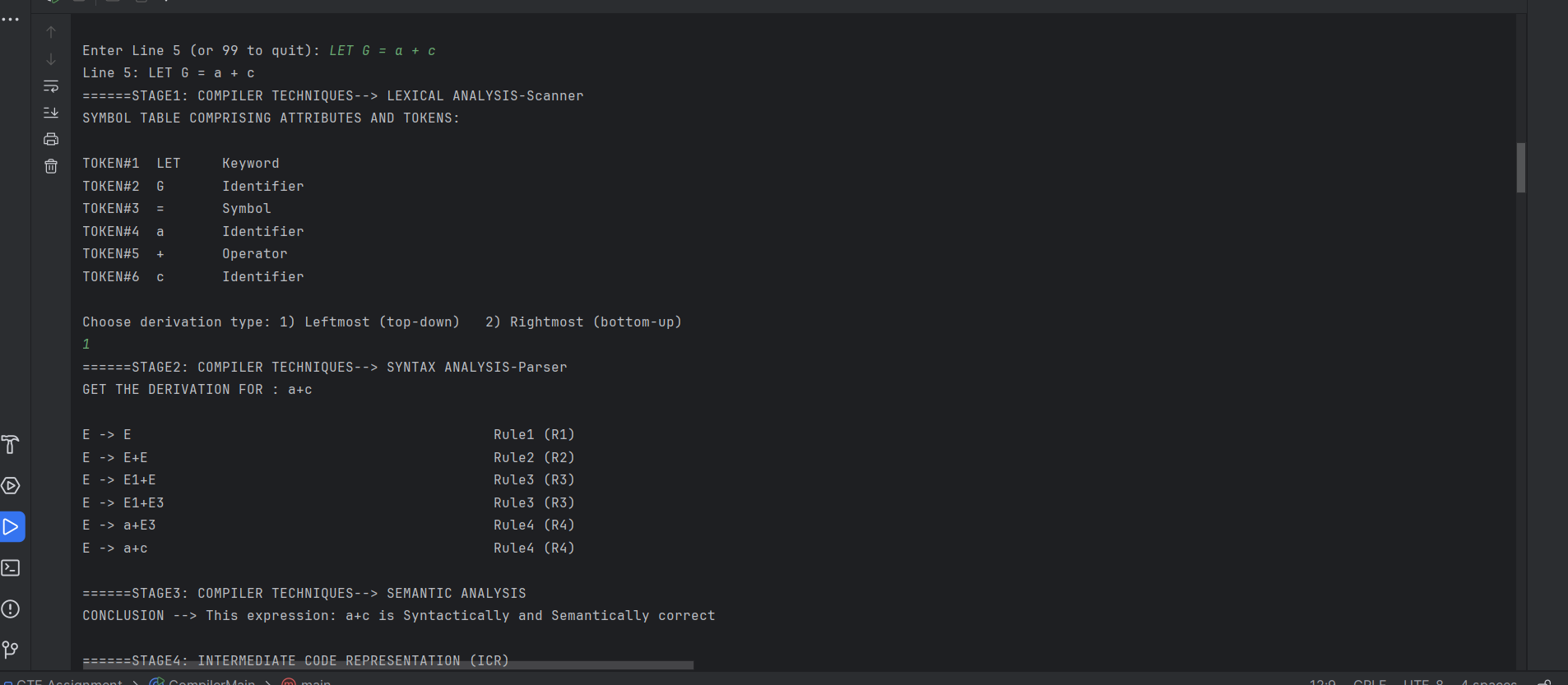
**SCREEN SHOTS OF RUNNING THE PROGRAM**

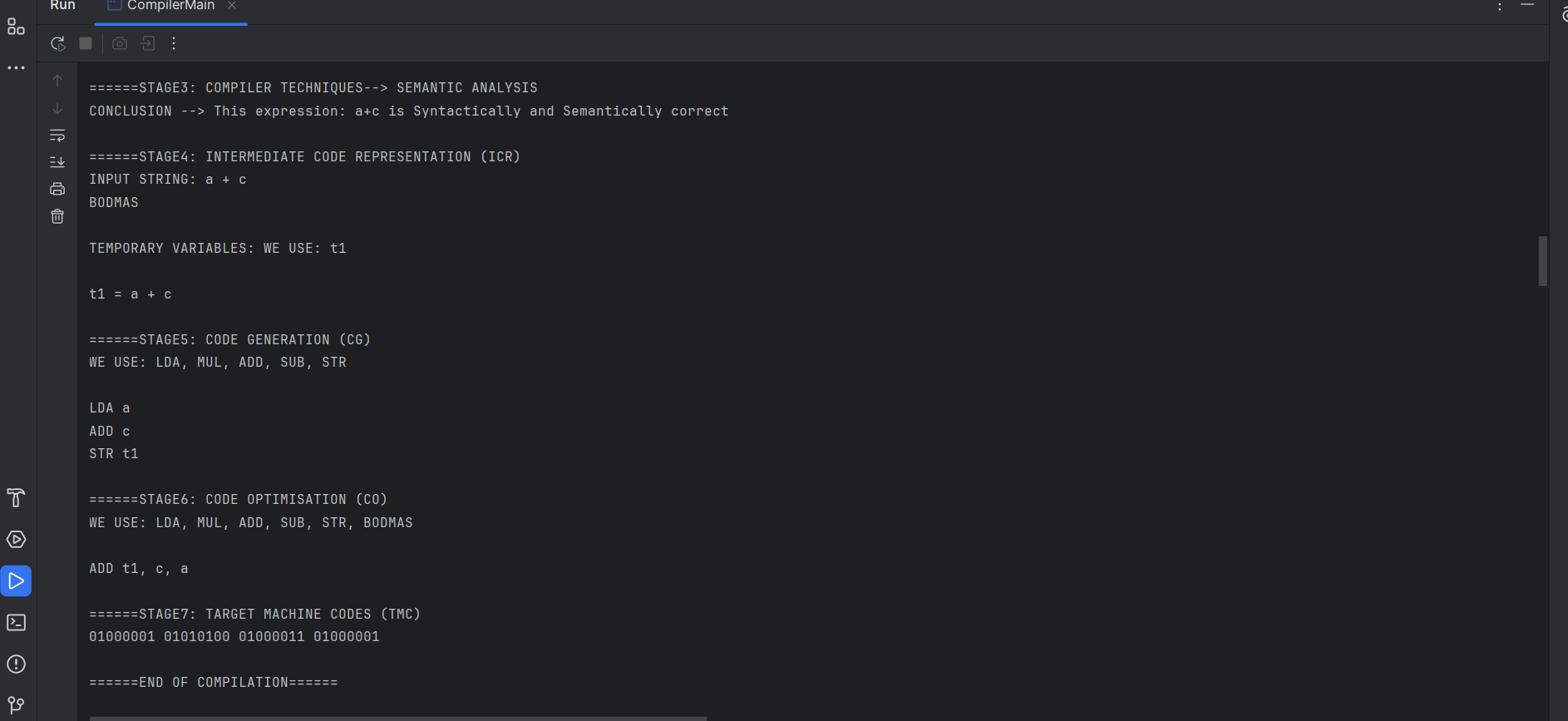
1. **Iterative**

****

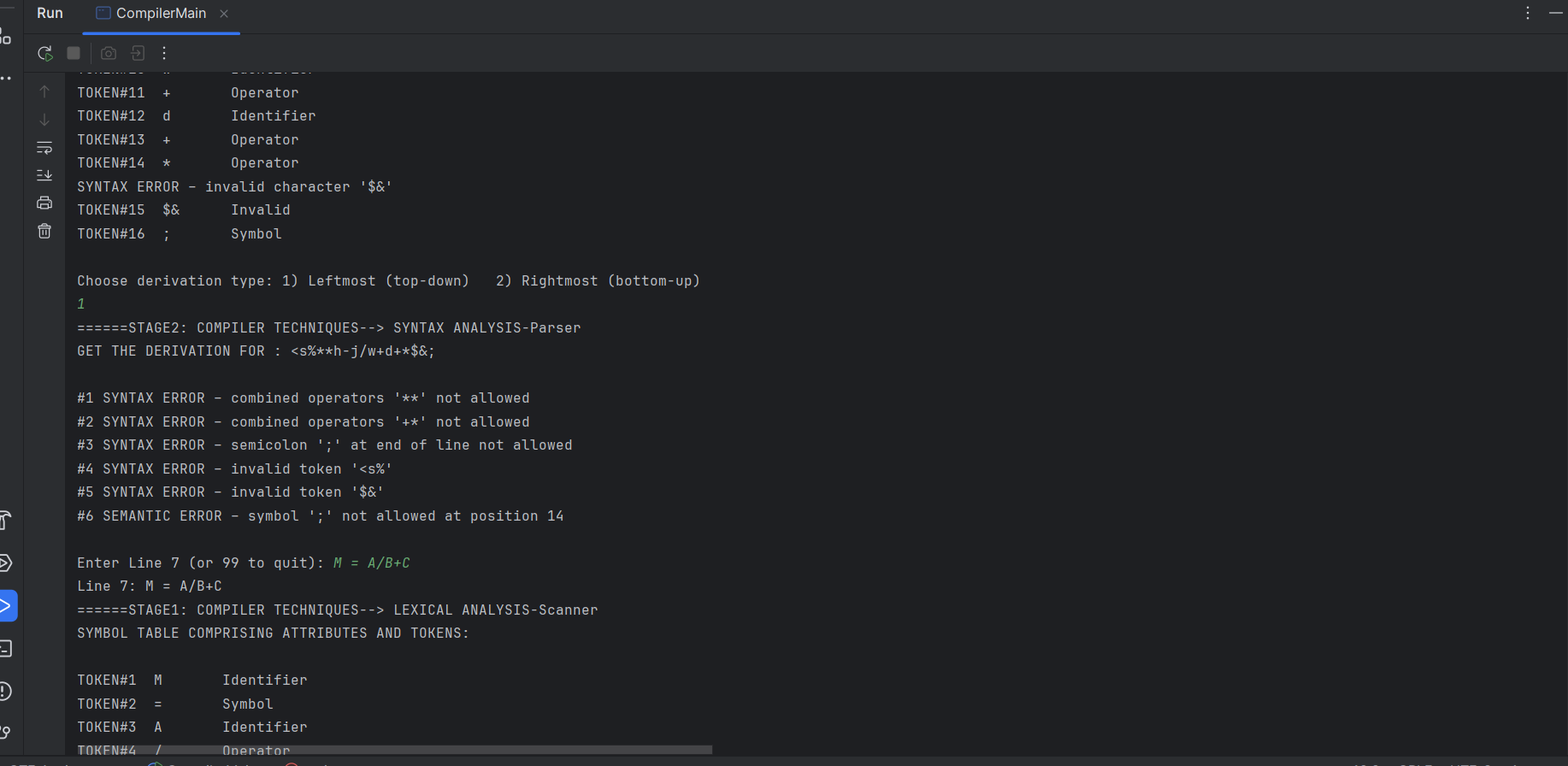
****

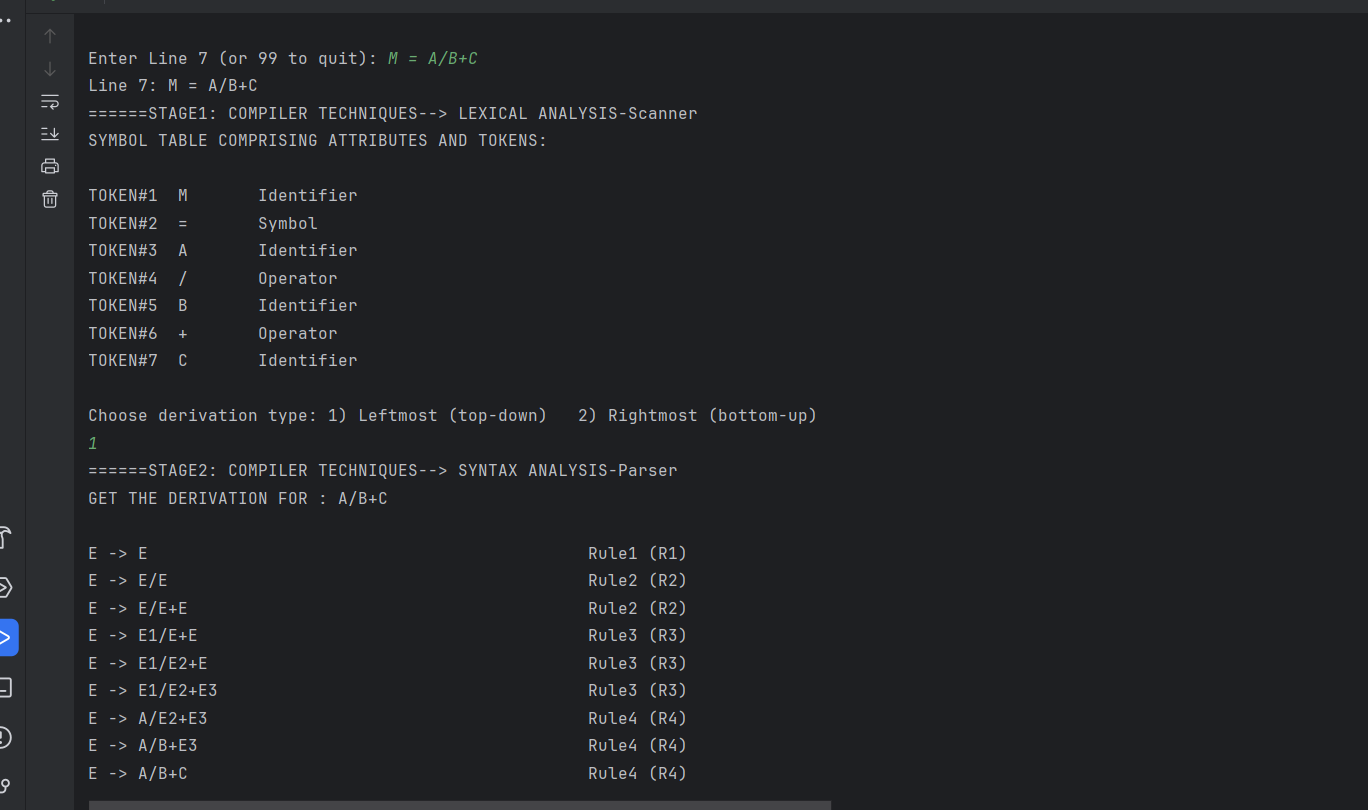
****

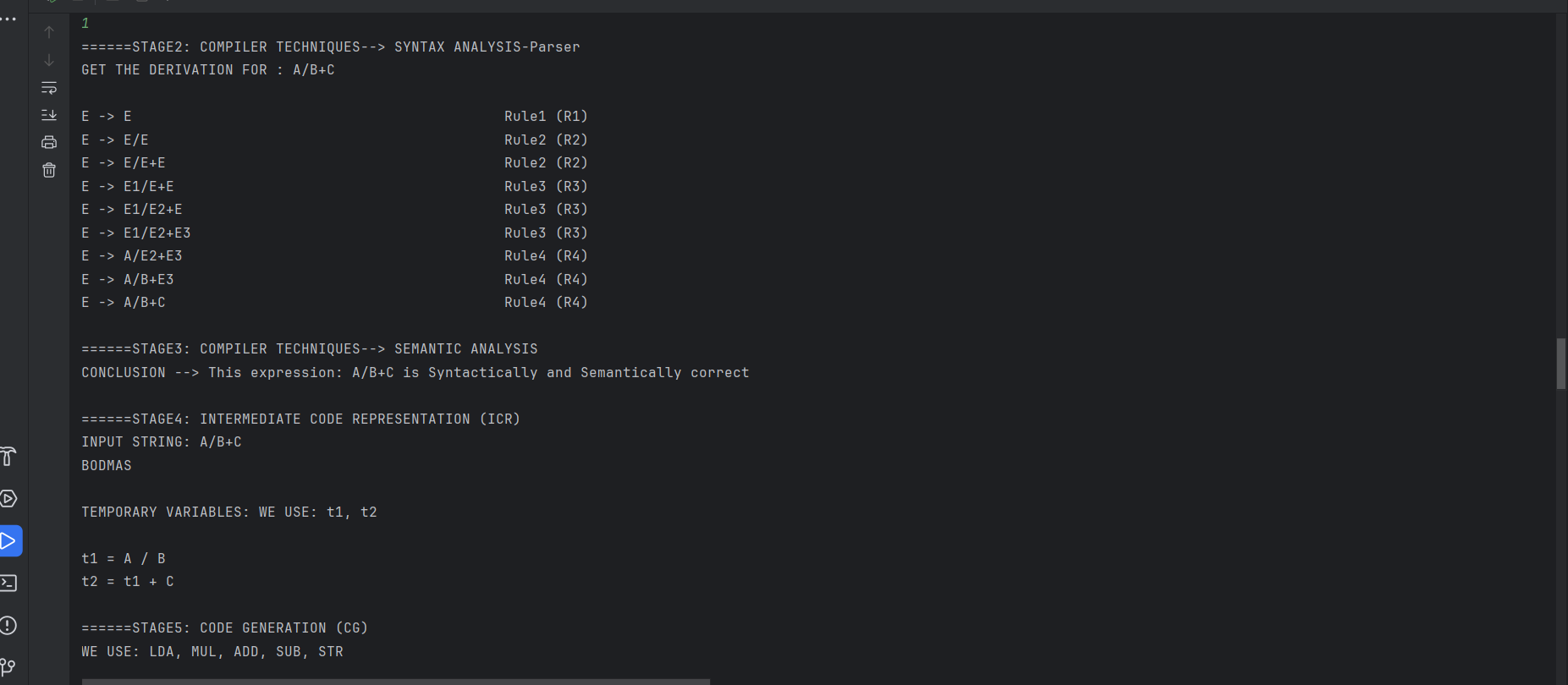
****

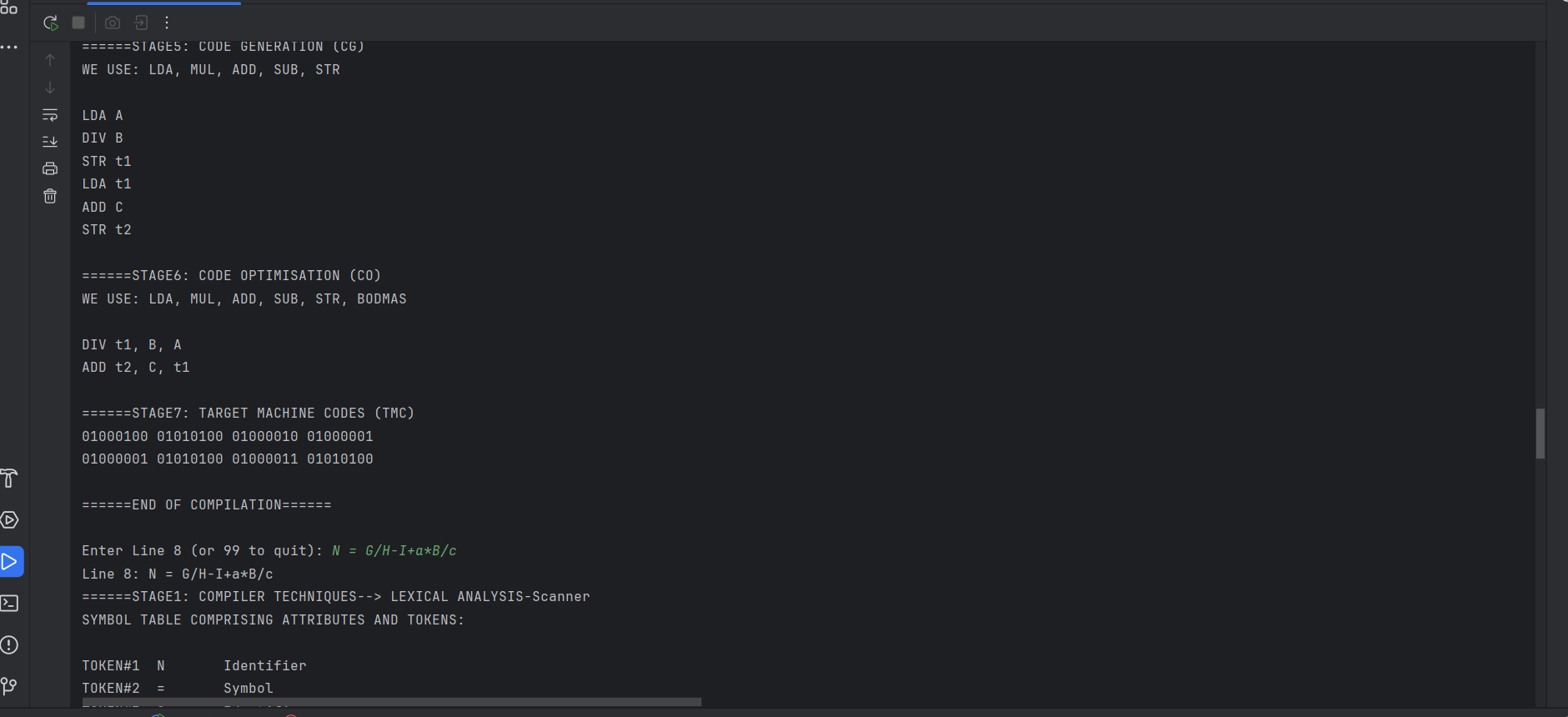
****

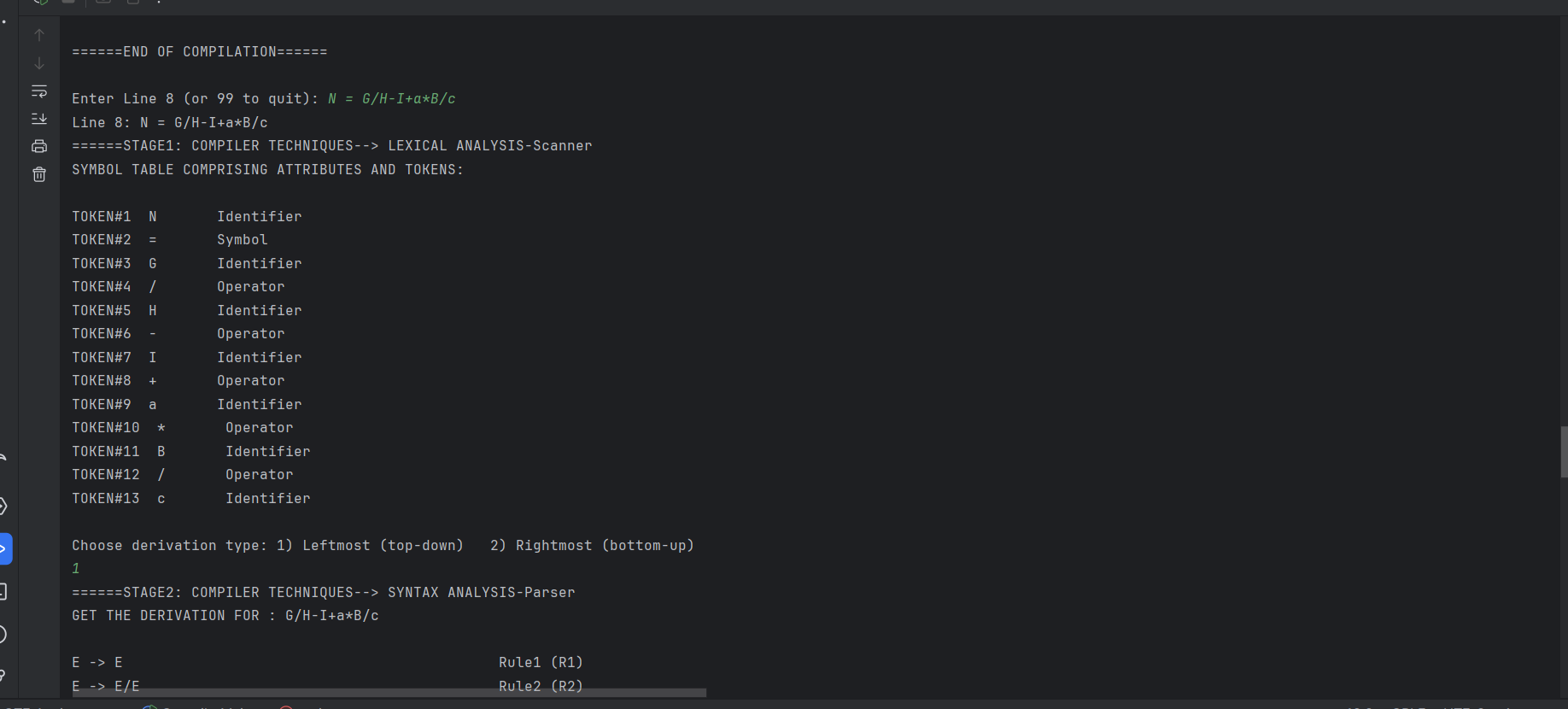
****

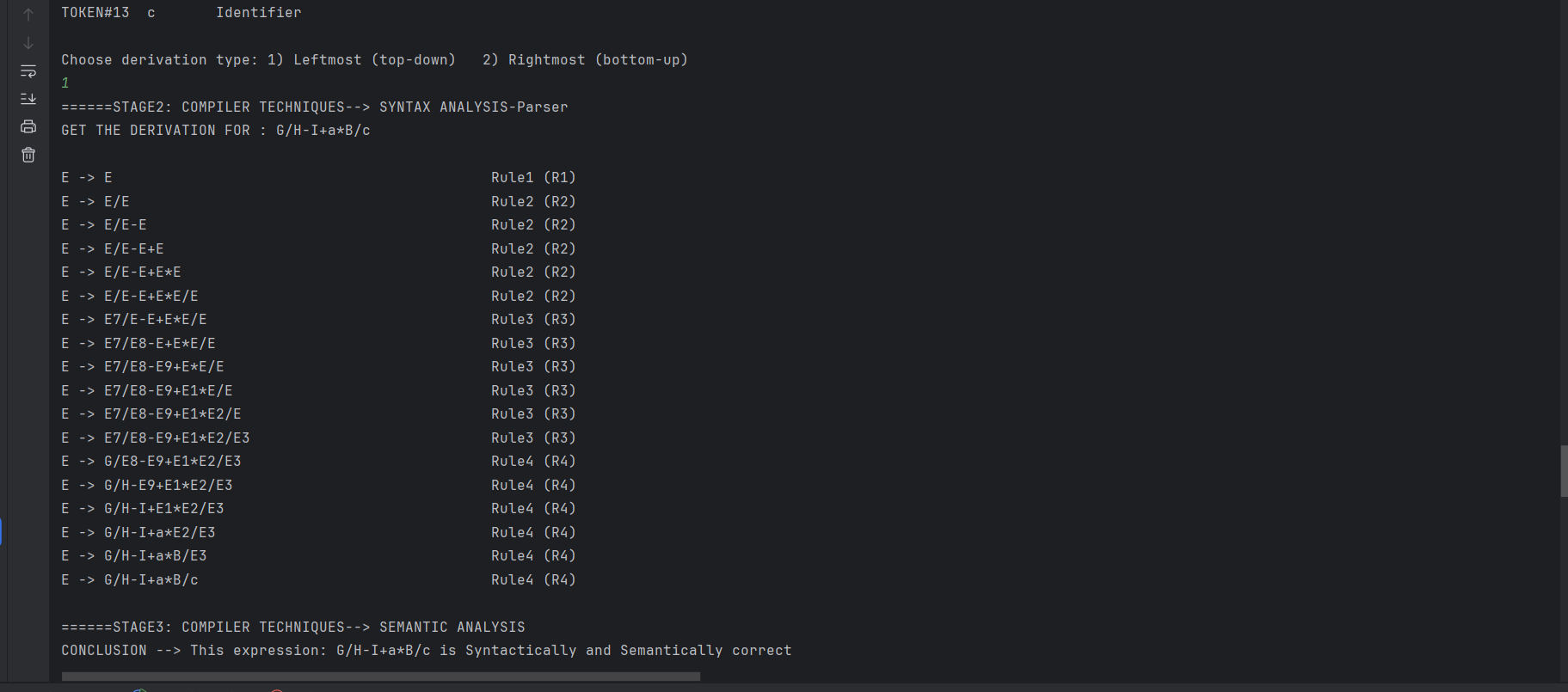
****

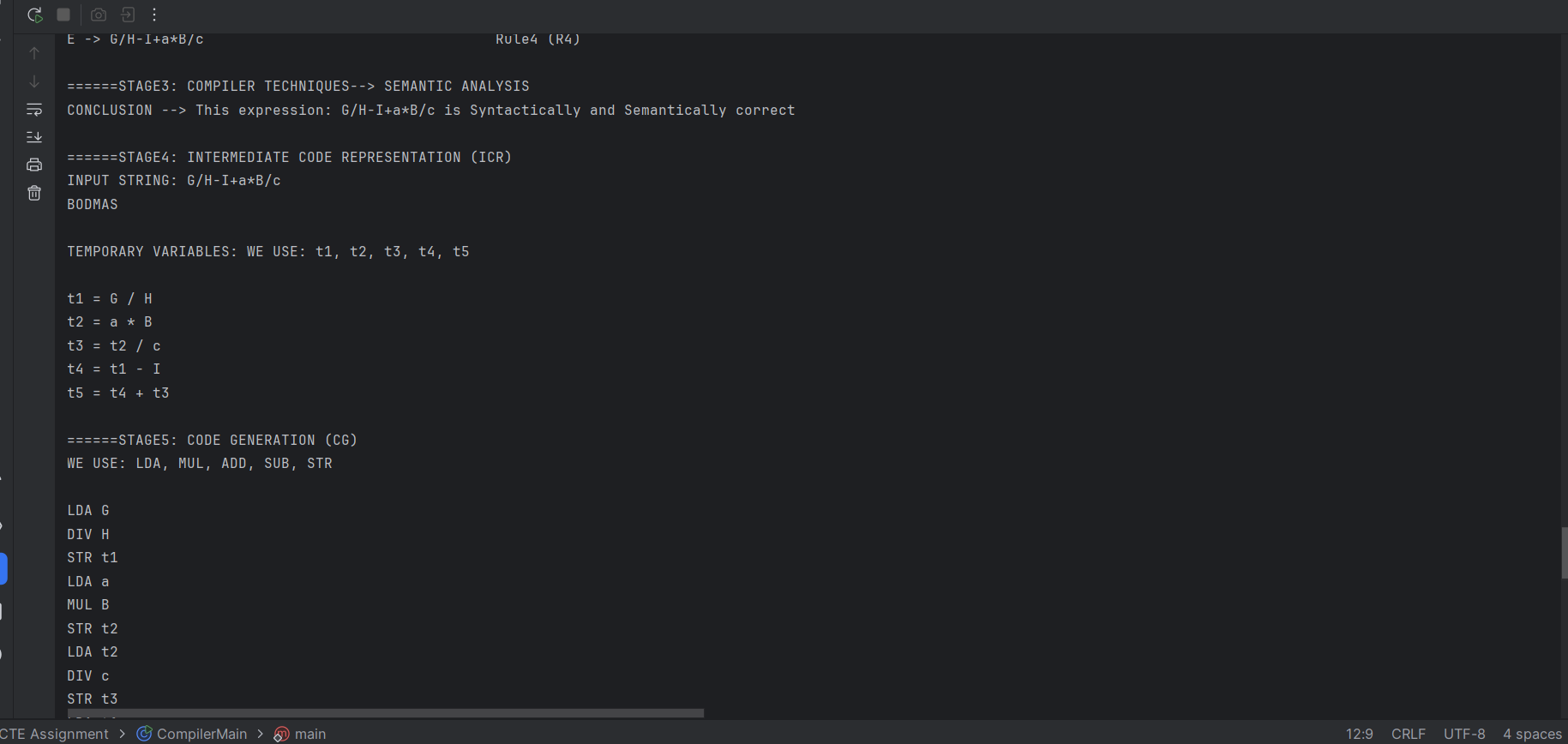
****

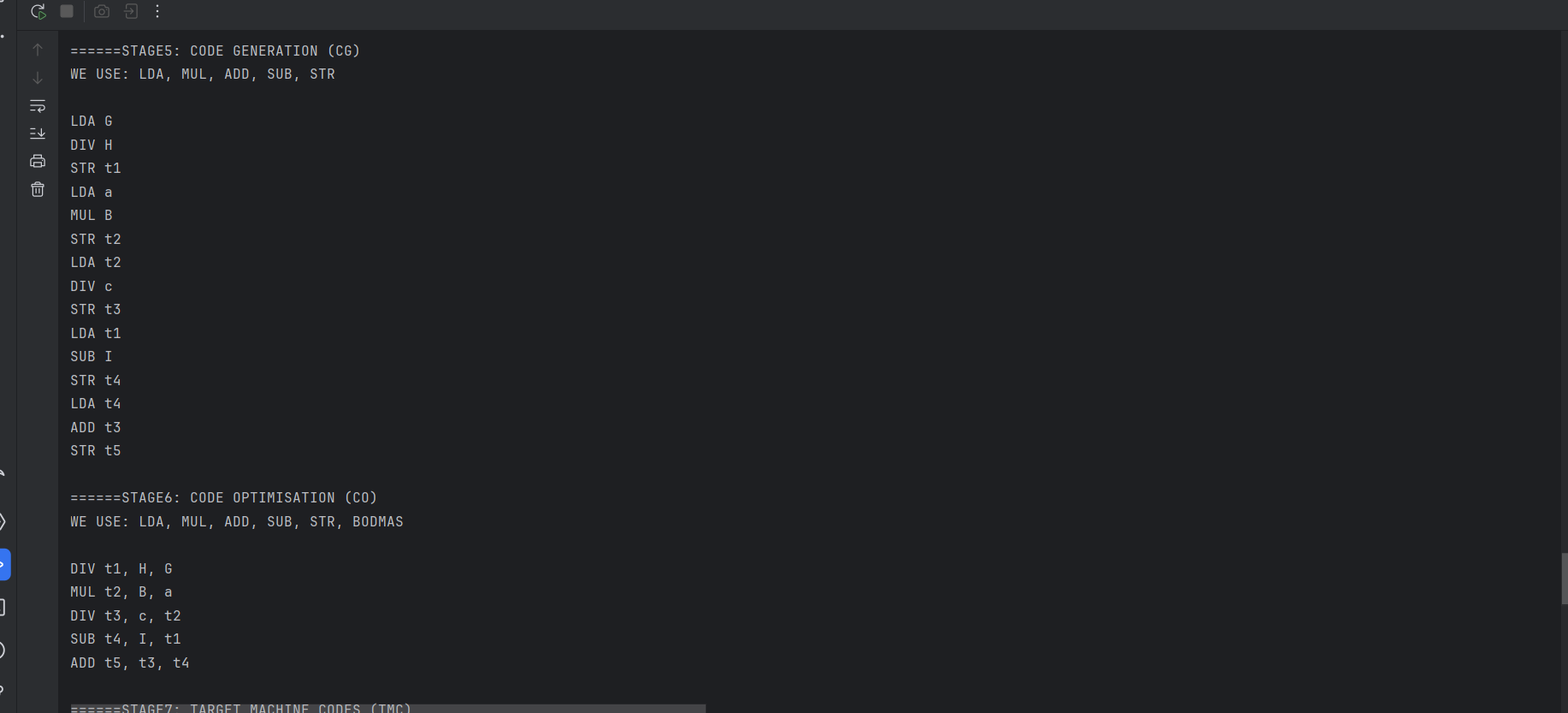
****

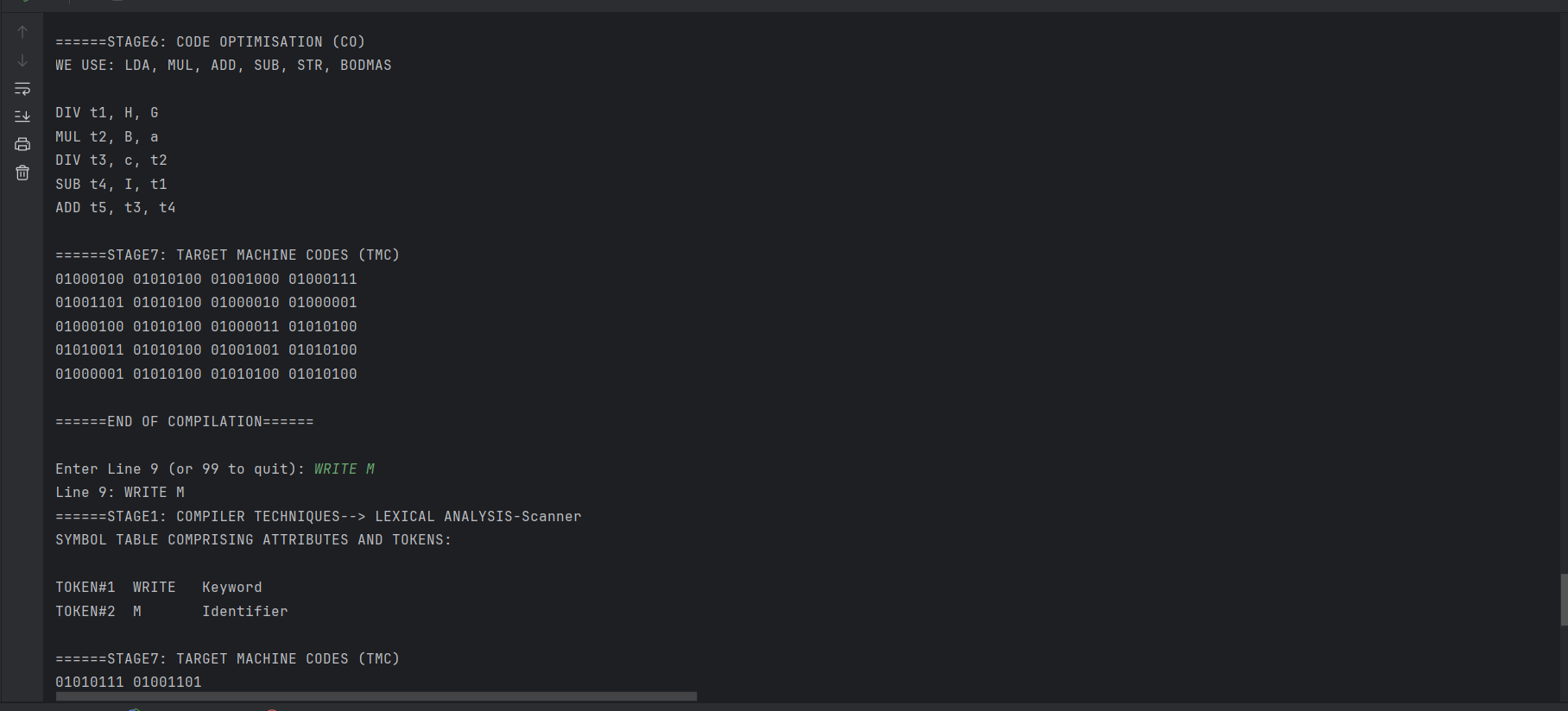
****

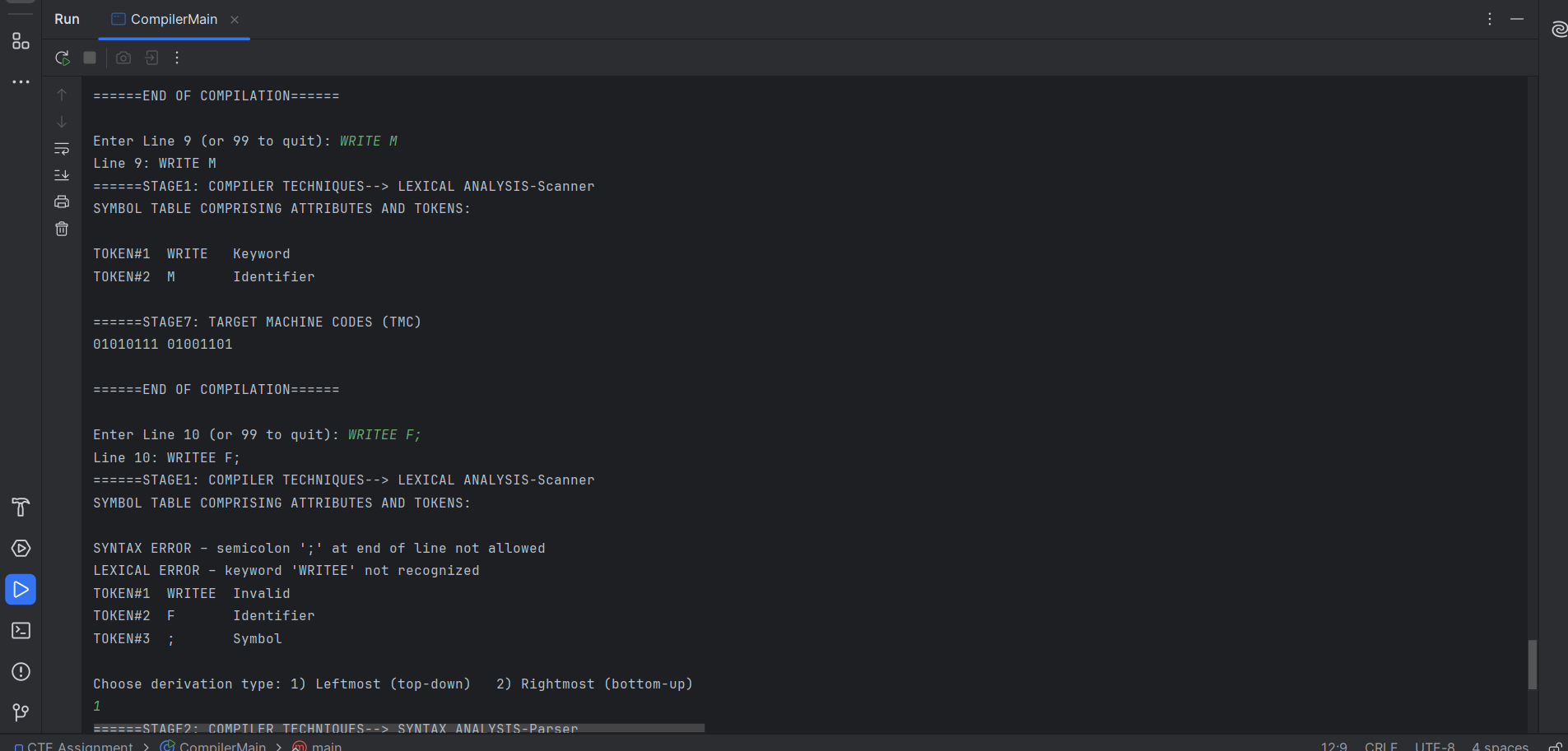
****

****

****

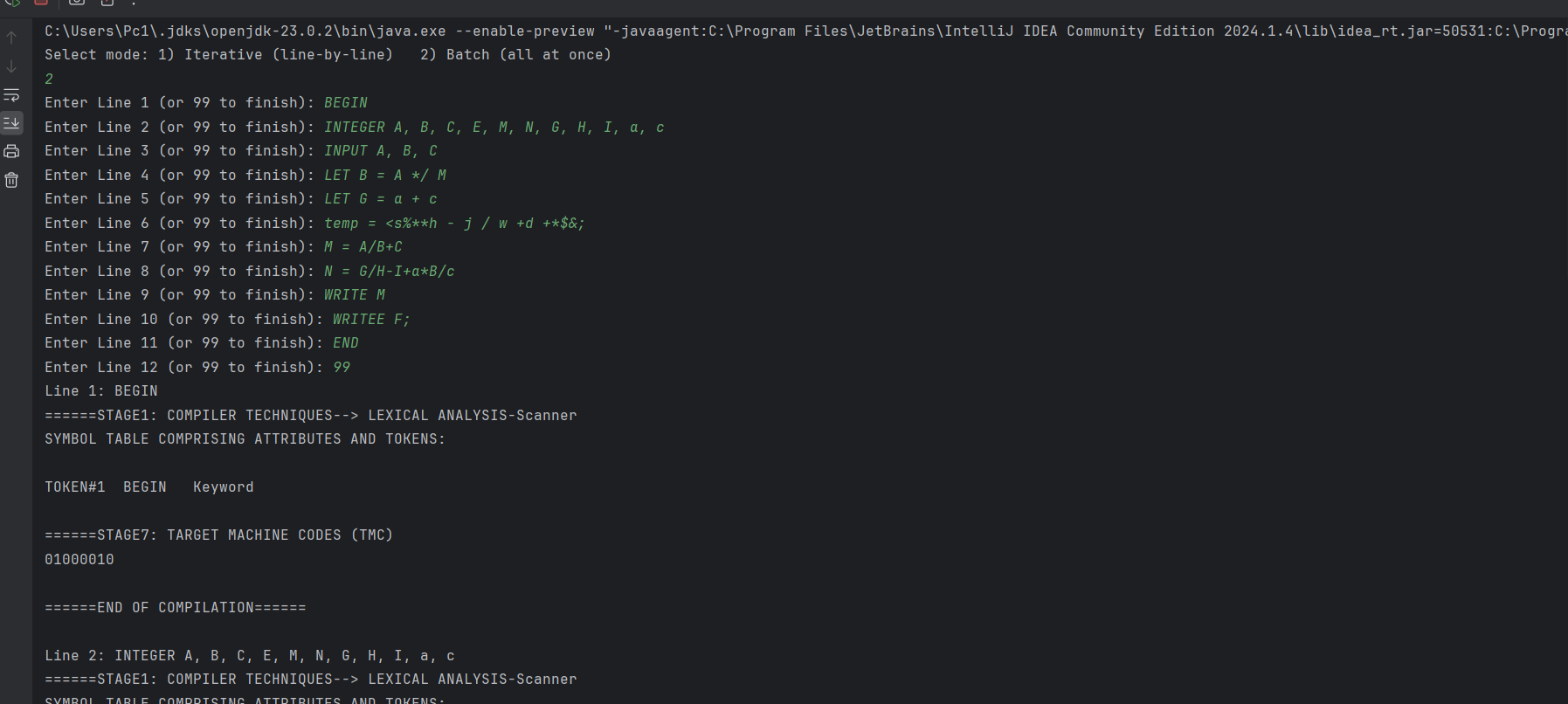
****

****

****

****

1. **Batch**

****

**Same Output as Iterative**

**ROLE PLAYED BY EACH TEAM MEMBERS**

|  |  |  |  |
| --- | --- | --- | --- |
| **Sn** | **Name** | **Student Number** | **Role played in the project** |
| 1 | Mario Uushunga | 223023795 | Worked on Derivation (Stage 2), Worked on Intermediate Representation (Stage 4) |
| 2 | Prince Itope | 223102792 | Worked on Lexical Analyzer (Stage 1), Worked on Target Machine Code (Stage 7) |
| 3 | Diago Flavio De Oliveira | 223086525 | Worked on Code Generation (Stage 5), Worked on Code Optimization (Stage 6) |
| 4 | Benjamin Nehoya | 223056553 | Worked on Sematic Analysis (Stage 3), Worked on Syntax Analysis (Stage 2) i.e Finding and catching errors |
| 5 | Gospel Nwagbara | 223116866 | Worked on Target Machine Code (Stage 7), Worked on Lexical Analyzer (Stage 1) |

***This Table should be filled by the Team Leader (TL)***

FT and PT Students are to take note of the following:

* Groups/Teams should edit this MS Word Template to suit their Group Project Assignment Topic.
* The Assignment should be done as a group, but each student/group member must submit the MS Word file on elearning and also send the file to the email: [postgraduatementor@gmail.com](mailto:postgraduatementor@gmail.com) on or before 23h59 on Fri 2 May 2025. The reason for also sending to email is because some MS Word files may be too large to upload on elearning.
* The name of each MS Word file should be StudentSurname-StudentNumber (e.g. Steven-2322220911).
* Date of Group Assignment presentation/defence on FTF is as stated on the course outline: 5 – 9 May 2025.