**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** | **Specialization** | **Programme** |
| 1 | Edmund Jansen **(Team Leader)** | 223118796 | Software Development | Part-Time |
| 2 | Rachel Lazarus | 223001244 | Software Development | Part-Time |
| 3 | Keenan Husselmann | 214076784 | Software Development | Part-Time |
| 4 | Sander Santana | 223034738 | Software Development | Part-Time |
| 5 | Louisiito Cloete | 223000132 | Software Development | Part-Time |
| 6 | Aloys Mwashekele | 223073458 | Software Development | Part-Time |

**SUBMITTED BY :** Edmund Jansen

**TITLE OF PROJECT:**

Development of a Mini Compiler System Using Java

**DATE**:

02 May 2025

**ROLE PLAYED BY EACH TEAM MEMBERS**

|  |  |  |  |
| --- | --- | --- | --- |
| **Sn** | **Name** | **Student Number** | **Role played in the project** |
| 1 | Edmund Jansen **(TL)** | 223118796 | Did Documentation and Machine Target Code. |
| 2 | Rachel Lazarus | 223001244 | Wrote Syntax Analysis Code. |
| 3 | Keenan Husselmann | 214076784 | Wrote Lexical Analysis Code. |
| 4 | Sander Santana | 223034738 | Wrote Intermediate Code representation(ICR) code. |
| 5 | Louisiito Cloete | 223000132 | Wrote Code Generation code. |
| 6 | Aloys Mwashekele | 223073458 | Wrote Code Optimzation code. |

**THE JAVA PROGRAM SOURCE CODES:**

package compiler;

import java.util.ArrayList;

import java.util.List;

public class Compiler {

private static final String[] KEYWORDS = {"BEGIN", "INTEGER", "LET", "INPUT", "WRITE", "END"};

private static final String[] OPERATORS = {"+", "-", "/", "\*"};

private static final String[] SYMBOLS = {"="};

private static final String IDENTIFIER\_REGEX = "[a-zA-Z]";

public static void main(String[] args) {

String[] program = {

"BEGIN INTEGER A, B, C, E, M, N, G, H, I, a, c",

"INPUT A, B, C",

"LET B = A \*/ M",

"LET G = a + c",

"temp = <s%\*\*h - j / w +d +\*$&;",

"M = A/B+C",

"N = G/H-I+a\*B/c",

"WRITE M",

"WRITEE F;",

"END"

};

System.out.println("Line-by-Line Compilation:");

compileLineByLine(program);

System.out.println("\nAll-at-Once Compilation:");

compileAllAtOnce(program);

}

private static void compileLineByLine(String[] program) {

for (int i = 0; i < program.length; i++) {

System.out.println("\nLine " + (i + 1) + ": " + program[i]);

compileLine(program[i], i + 1);

}

}

private static void compileAllAtOnce(String[] program) {

StringBuilder fullProgram = new StringBuilder();

for (String line : program) {

fullProgram.append(line).append("\n");

}

System.out.println("\nFull Program:\n" + fullProgram.toString());

compileFullProgram(fullProgram.toString());

}

private static void compileLine(String line, int lineNumber) {

List<String> tokens = lexicalAnalysis(line, lineNumber);

if (tokens != null) {

syntaxAnalysis(tokens, lineNumber);

}

}

private static void compileFullProgram(String program) {

String[] lines = program.split("\n");

for (int i = 0; i < lines.length; i++) {

System.out.println("\nLine " + (i + 1) + ": " + lines[i]);

compileLine(lines[i], i + 1);

}

}

private static List<String> lexicalAnalysis(String line, int lineNumber) {

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

String[] parts = line.split("\\s+|(?=[=+\\-\*/,;])|(?<=[=+\\-\*/,;])");

for (String part : parts) {

part = part.trim();

if (!part.isEmpty()) {

tokens.add(part);

}

}

for (String token : tokens) {

if (isKeyword(token) || isIdentifier(token) || isOperator(token) || isSymbol(token)) {

// valid token

} else if (token.matches("[0-9]+")) {

System.out.println("Lexical Error at line " + lineNumber + ": Numbers are not allowed: " + token);

return null;

} else if (token.equals("WRITEE")){

System.out.println("Lexical Error at line " + lineNumber + ": Misspelled Keyword: "+ token);

return null;

} else if (token.contains("%")||token.contains("$")||token.contains("&")||token.contains("<")||token.contains(">")){

System.out.println("Lexical Error at line " + lineNumber + ": Invalid Symbol: "+ token);

return null;

} else {

System.out.println("Lexical Error at line " + lineNumber + ": Invalid token: " + token);

return null;

}

}

return tokens;

}

private static void syntaxAnalysis(List<String> tokens, int lineNumber) {

for (int i = 0; i < tokens.size() - 1; i++) {

if (isOperator(tokens.get(i)) && isOperator(tokens.get(i + 1))) {

System.out.println("Syntax Error at line " + lineNumber + ": Two consecutive operators: " + tokens.get(i) + " " + tokens.get(i + 1));

return;

}

}

if (tokens.get(tokens.size()-1).equals(";")){

System.out.println("Syntax Error at line "+ lineNumber + ": Semicolon at end of line not allowed");

return;

}

if (lineNumber == 4 || lineNumber == 6 || lineNumber == 7) {

semanticAnalysis(tokens, lineNumber);

}

}

private static void semanticAnalysis(List<String> tokens, int lineNumber) {

// Simple example, can be extended for more complex semantic checks

System.out.println("Semantic Analysis passed for line " + lineNumber);

intermediateCodeGeneration(tokens, lineNumber);

}

private static void intermediateCodeGeneration(List<String> tokens, int lineNumber) {

System.out.println("Intermediate Code Generation for line " + lineNumber + ": " + tokens);

optimization(tokens, lineNumber);

}

private static void optimization(List<String> tokens, int lineNumber) {

System.out.println("Optimization for line " + lineNumber + ": " + tokens);

codeGeneration(tokens, lineNumber);

}

private static void codeGeneration(List<String> tokens, int lineNumber) {

System.out.println("Code Generation for line " + lineNumber + ": " + tokens);

}

private static boolean isKeyword(String token) {

for (String keyword : KEYWORDS) {

if (keyword.equals(token)) {

return true;

}

}

return false;

}

private static boolean isIdentifier(String token) {

return token.matches(IDENTIFIER\_REGEX);

}

private static boolean isOperator(String token) {

for (String operator : OPERATORS) {

if (operator.equals(token)) {

return true;

}

}

return false;

}

private static boolean isSymbol(String token) {

for (String symbol : SYMBOLS) {

if (symbol.equals(token)) {

return true;

}

}

return false;

}

}

**Screenshots of running codes:**

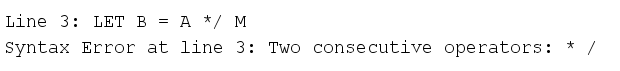


A close-up of a sign

AI-generated content may be incorrect.

A black text on a white background

AI-generated content may be incorrect.

A black text on a white background

AI-generated content may be incorrect.

A close-up of a number

AI-generated content may be incorrect.

A black text on a white background

AI-generated content may be incorrect.

A black text on a white background

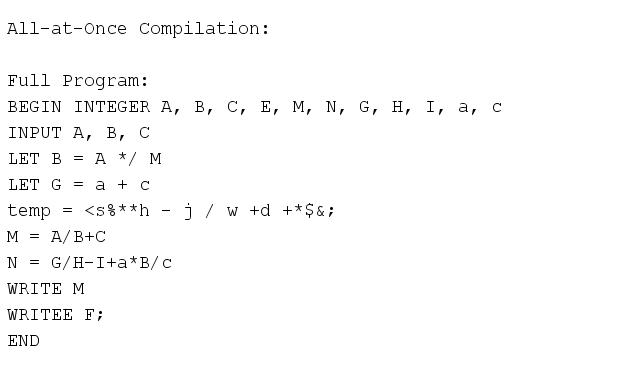
AI-generated content may be incorrect.

A close up of a sign

AI-generated content may be incorrect.

A black text on a white background

AI-generated content may be incorrect.



**Conclusion**

While our mini compiler might have limitations and areas for future expansion (such as handling more complex language features, generating optimized code, or targeting a specific virtual machine), we are proud of the progress we've made. This project serves as a solid foundation for further exploration into the fascinating world of compilers and programming language implementation. We recognize that building a fully-fledged compiler is a monumental task, and this experience has given us a newfound respect for the complexity and ingenuity behind the tools we use every day as programmers.