

Language Implementation of a Grammar of Sentences in Propositional Logic

A Project Documentation

Arriola
Bisenio
Domingo
Pagunsan
Zamora

Introduction

Backus Naur Form (BNF)—a simple yet powerful metalanguage which uses abstractions to describe logic—became the standard of describing programming language syntax after developments in human language classes by Noam Chomsky, accompanied by groundbreaking propositions by John Backus and Peter Naur in 1959 and 1960 (Sebesta, 2012).

To appreciate BNF as a foundation for studying programming languages, this project was built on the premise of implementing the BNF grammar using a contemporary language, specifically, Java.

The project is currently limited to three identifiers: P, Q, and S, for a possible maximum of eight rows in a truth table.

Table of contents

Introduction	2
01 Overview of Program Design	4
1.1 Main driver Scan user input	4
1.2 The LogicScanner	4
1.3 Token and TokenType classes	6
1.4 The Parser	7
1.5 The Evaluator	8
1.6 The Error Handler	10
02 Building and executing the program	11
2.1 Download the source code as a .ZIP file	11
2.2 Extract the contents of the .ZIP file	11
2.3 Locate and open "run.bat"	12
2.4 Input "LOGIC sentence.pl"	12
2.5 Editing sentence.pl	13
03 Difficulties encountered	14

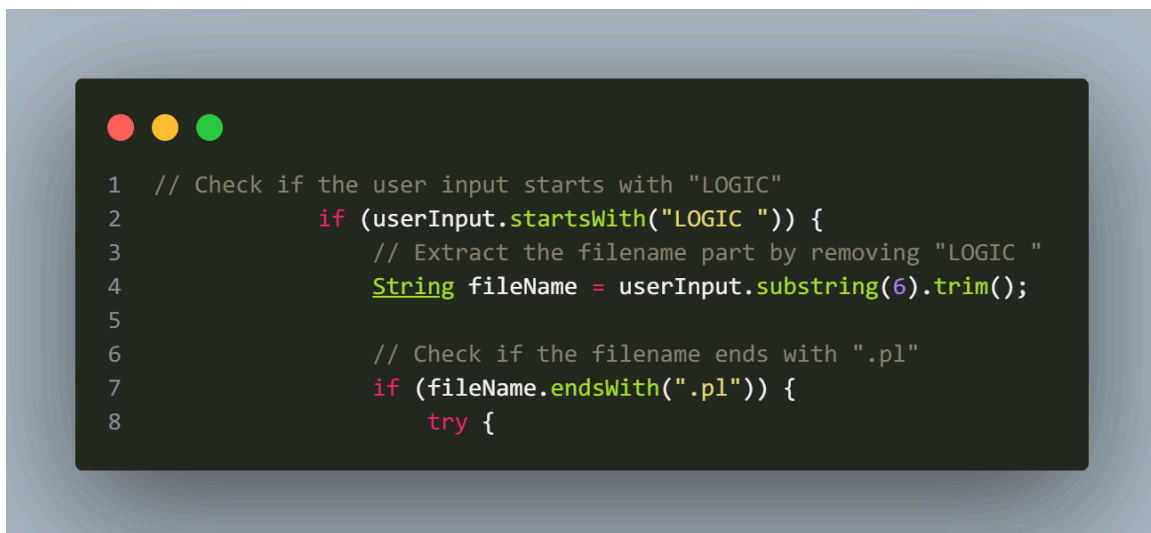
01 Overview of Program Design

A Java implementation of BNF takes advantage of Java's object-oriented nature. Hence, the project is split into seven java files, each serving a specific purpose, namely, in order of function: **(1)** the Main driver class, **(2)** a Logic Scanner, **(3)** classes for handling Tokens, **(4)** a Parser, **(5)** an Evaluator, and **(6)** an Error Handler.

1.1 Main driver | Scan user input

The main driver prompts for user input. Only "LOGIC <filename>.pl" is accepted by the program, and will display error messages accordingly. This is implemented using try-catch statements.

By default, the repository includes a pre-made "sentence.pl" file.



A code snippet showing the if-statements which check for user input and file name.

Once valid user input is accepted, the contents of the .pl file are passed on to the **LogicScanner** method.

1.2 The LogicScanner

The **LogicScanner** class receives a String input from the "sentence.pl" file through the Main driver (see Section 1.7), and "tokenizes" the input into four definitions inside the grammar: **keywords**, **identifiers**, **literals**, and **symbols**.

```

1 private static final String[] KEYWORDS = {"NOT", "AND", "OR", "IMPLIES", "EQUIVALENT"};
2
3 // Identifiers
4 private static final String[] IDENTIFIERS = {"P", "Q", "S"};
5
6 // Booleans
7 private static final String[] LITERALS = {"TRUE", "FALSE"};
8
9 // Parentheses
10 private static final char[] SYMBOLS = {'(', ')'};

```

The keywords, identifiers, literals, and symbols are defined in the beginning of the LogicScanner Class.

```

1 public List<Token> tokenize(String input) {
2
3     // ArrayList for tokens
4     // Use ArrayList instead of Arrays for flexibility in array size
5     List<Token> tokens = new ArrayList<>();

```

The tokenize method which takes a String input from the main driver class.

For the classification and addition of tokens as the **LogicScanner** processes the input from the .pl file, the helper method **classifyAndAddToken** runs through a series of if-else-if statements.

```

1 if (isKeyword(token)) { // "NOT", "AND", "OR", "IMPLIES", "EQUIVALENT"
2     tokens.add(new Token(token, TokenType.KEYWORD));
3 } else if (isIdentifier(token)) { // "P", "Q", "S"
4     tokens.add(new Token(token, TokenType.IDENTIFIER));
5 } else if (isLiteral(token)) { // "TRUE", "FALSE"
6     tokens.add(new Token(token, TokenType.LITERAL));
7 } else { // Anything other than those mentioned
8     tokens.add(new Token(token, TokenType.UNKNOWN)); // Catch invalid tokens
9 }
10 }

```

Tokens are classified as keyword, identifier, literal, or in case of error, unknown.

For clarity, the methods for displaying token values and types are abstracted into the **Token** and **TokenType** classes.

1.3 Token and TokenType classes

In the **Token** class, the value and type of tokens are declared as variables alongside constructors and getters as well as a method to display the processed token values and types.

```
1 // Value of the token: "P", "TRUE", etc.
2 private String value;
3 // Type of the token: "IDENTIFIER", "SYMBOL", etc.
4 private TokenType type;
```

```
1 // Constructor for the Token
2 public Token(String value, TokenType type) {
3     this.value = value;
4     this.type = type;
5 }
```

Declaration of token value and type and the respective constructor.

```
1 public void displayToken() {
2     System.out.println("Token Value: " + value + ", Token Type: " + type);
3 }
```

Method that prints the processed Token Value and Type.

The **TokenType** class simply contains an enum of Token Types.

```
1 public enum TokenType {
2     KEYWORD,      // "NOT", "AND", "OR", "IMPLIES", "EQUIVALENT"
3     IDENTIFIER,   // "P", "Q", "S"
4     LITERAL,      // "TRUE", "FALSE"
5     SYMBOL,       // "(", ")"
6     UNKNOWN       // For invalid tokens
7 }
```

A code snippet showing the enum TokenType inside the **TokenType** class.

After the entire String input from the .pl file is tokenized, the Main driver class proceeds to the **Parser**.

1.4 The Parser

After receiving token inputs from the **Main** driver, **Parser** verifies the grammar of the tokens processed from the .pl file.

```
1 while (tokenIndexValue < forTokens.size() && currentToken().getType() == TokenType.KEYWORD) {
2     String connective = currentToken().getValue(); // Gets the connective
3     consumeToken(); // Consumes the connective
4     Sentence right = parseAtomicOrComplex(); // Parses the right side of the sentence
5     left = new ComplexSentence(left,connective,right); // Constructs a new ComplexSentence with the left and right parts
6 }
```

A while loop constructs a ComplexSentence as it iterates through the tokens passed by the Main driver class.

The **Parser** catches both unexpected and missing tokens, and throws SyntaxError messages accordingly.

```
1 if (tokenIndexValue < forTokens.size()) {
2     throw new SyntaxError("\n Unexpected token: " + currentToken().getValue());
3 }
4 return sentence;
5 }
```

Several if statements throw an "Unexpected token" message when there are errors.

Aside from dealing with the basic atomic sentences, the **Parser** is able to handle compound sentences which involve parentheses, e.g. "P AND (Q OR S)".

```
1 if (token.getType()
2     == TokenType.SYMBOL
3     && token.getValue().equals("(")) {
4     consumeToken(); // For "(" token
5     Sentence innerSentence = parseSentence(); // Parses the inner sentence
```

Upon the detection of parentheses, the **Parser** parses an inner sentence.

A complete sentence is returned to the Main driver, which then calls the **Evaluator**.

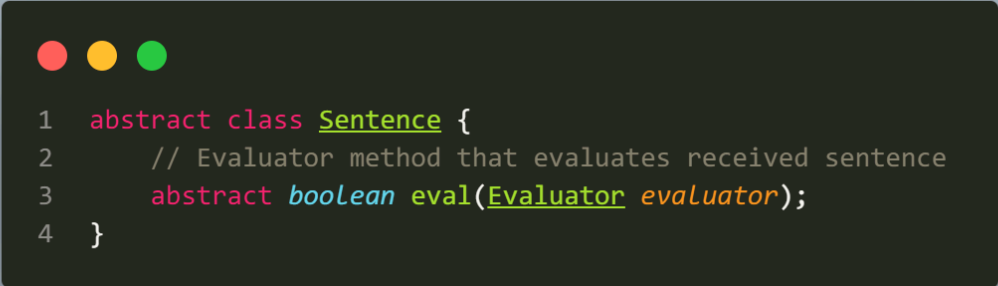
1.5 The Evaluator

To complete the basic logical process, the **Evaluator** processes the parsed sentence and creates a standard Truth Table.

The number of rows of the Truth Table depend on the number of identifiers, i.e. for 1 identifier there are 2 rows, for 2 identifiers there are 4 rows, and for 3 identifiers, there are 8 rows.

As mentioned in the introduction, the project handles up to 3 identifiers only.

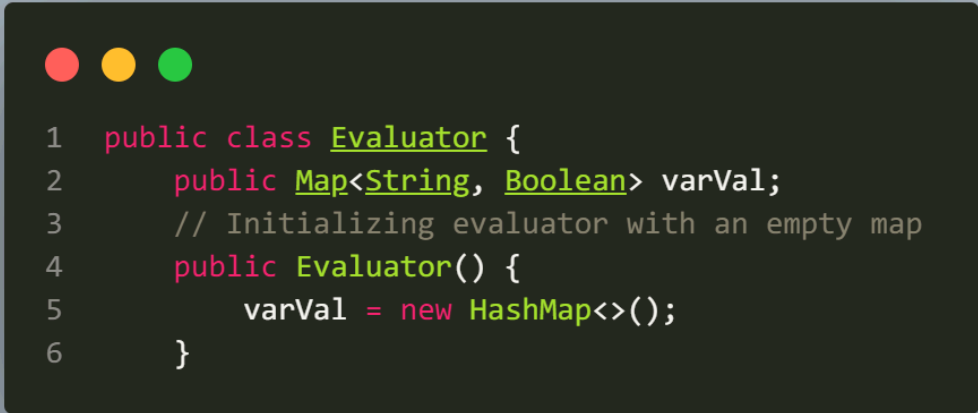
To represent all possible forms of sentences i.e. Atomic Sentences and Complex Sentences, the abstract **Sentence** class comprises several subclasses which handle truth values as boolean data.



```
1  abstract class Sentence {
2      // Evaluator method that evaluates received sentence
3      abstract boolean eval(Evaluator evaluator);
4  }
```

The superclass **Sentence** for Logical sentences. It is inherited from by **AtomicSentence**, **Variable**, **ComplexSentence**, and **Not**, which are specific logical forms that entail different truth values.


After handling the boolean properties of the parsed sentence from the **Main** driver class, the **Evaluator** class uses a Map to store the truth values of variables and sentences.



```
1  public class Evaluator {
2      public Map<String, Boolean> varVal;
3      // Initializing evaluator with an empty map
4      public Evaluator() {
5          varVal = new HashMap<>();
6      }
```

The **Map** object is responsible for mapping keys, i.e. TRUE or FALSE, to the values passed by the **Parser**.


Specifically, the `setvarVal` method sets the truth value for each variable.



```
1 public void setvarVal(String var, Boolean val) {
2     varVal.put(var, val);
3 }
```


A string variable and a boolean truth value are passed into the method.

Based on the information passed by the **Sentence** class, a boolean truth value is evaluated.



```
1 public boolean evaluate(Sentence sentence) {
2     return sentence.eval(this);
3 }
```

Finally, through nested **for** loops, with the **Sentence** class-provided sentences and variables as inputs, the **evaluateTable** method generates a truth table.



```
1 public void evaluateTable(Sentence sentence, String[] variables) {
2     int varNum = variables.length; // Number of variables (n)
3     int combNum = (int) Math.pow(2, varNum); // Number of combinations of variables (2^n)
4
5     for(int i=0; i<combNum; i++) {
6         for(int j=0; j<varNum; j++) {
7             // Calculates truth value for the variable (0 = false, 1 = true)
8             boolean value = ((i >> (varNum - 1 - j)) & 1) == 0;
9             setvarVal(variables[j], value);
10        }
```

1.6 The Error Handler

As the **Main** driver class reads the content of the .pl file through its helper **readFile** function, a while loop which lasts as long as the .pl file is being read, i.e. there are still lines for the program to read, checks for syntax errors through the **ErrorHandling** class.

The **ErrorHandling** method calls a **LogicScanner** instance and uses a boolean **errorCheck** method to iterate through each word in the sentence being read from the .pl file.



```
1  LogicScanner logScan = new LogicScanner();
2
3  public boolean errorCheck(String input) {
4      String[] container = spaceWords(input).split(" ");
5      List<String> wordContainer = new LinkedList<String> (Arrays.asList(container));
6
7      int wcN = wordContainer.size();
8      int count = 0;
9
10     for (int i = 0; i < wcN; i++) {
```

For loop logic is used to iterate through the **List** of words in the sentence.

The **ErrorHandling** method checks for the following:

- Whether the current word is valid
- Whether the current word and adjacent word is a keyword, excluding "NOT"
- Whether the current word and adjacent word are identifiers
- Whether the starting word is a keyword
- Whether the last word is a keyword

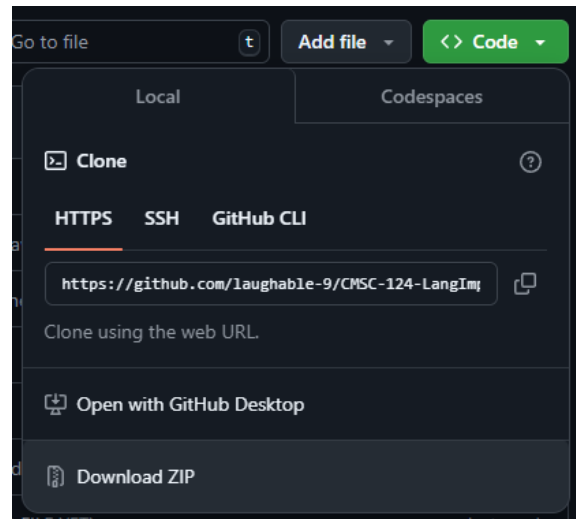
In the following section, instructions are provided for any user to be able to download and run this project on their computer.

02 Building and executing the program

To run this Java implementation of BNF, do the following steps:

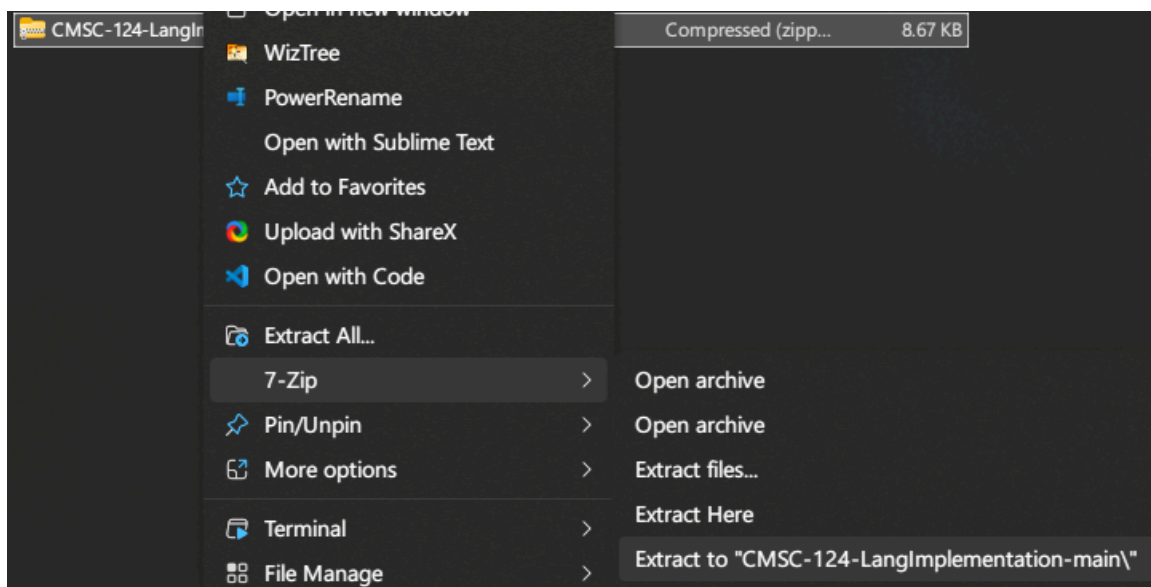
2.1 Download the source code as a .ZIP file

Open [this repository](#) and click on “Code” > “Download ZIP”.



2.2 Extract the contents of the .ZIP file

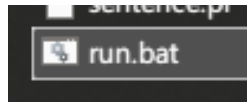
Extract the contents of “CMSC-124-LangImplementation-main.zip” **into a new folder**.



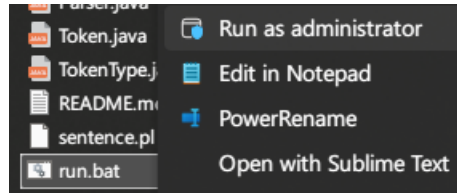
If you mistakenly extracted the files into a preexisting folder with unrelated files, it is advisable to create a new folder and move the contents of the .ZIP file into the new folder.

2.3 Locate and open “run.bat”

Look for the batch file, ‘run.bat’ inside the folder. Double click it and a command prompt window will appear.



If this fails, this is most likely due to the SmartScreen security feature installed in most present-day Windows systems. Run it as an administrator to bypass the issue.



2.4 Input “LOGIC sentence.pl”

Once you have successfully ran the batch file, this should be what appears on your command prompt window.

```
CMSC 124 - Language Implementation of a Grammar of Sentences in Propositional Logic
A Machine Project
Prepared by Arriola, Bisenio, Domingo, Pagunsan, Zamora
Compiling Java files.
ECHO is off.
Running...
Enter command: LOGIC sentence.pl
```

You will be prompted to enter a command. The program only accepts the strict format of ‘LOGIC <filename>.pl’. With the default ‘sentence.pl’ file that is included in the folder, you should be able to run the program successfully by inputting the following:

LOGIC sentence.pl

A truth table should now appear on your command prompt screen.

```
Enter command: LOGIC sentence.pl

Tokens:
Token Value: NOT, Token Type: KEYWORD
Token Value: (, Token Type: SYMBOL
Token Value: P, Token Type: IDENTIFIER
Token Value: AND, Token Type: KEYWORD
Token Value: (, Token Type: SYMBOL
Token Value: Q, Token Type: IDENTIFIER
Token Value: OR, Token Type: KEYWORD
Token Value: S, Token Type: IDENTIFIER
Token Value: ), Token Type: SYMBOL
Token Value: ), Token Type: SYMBOL

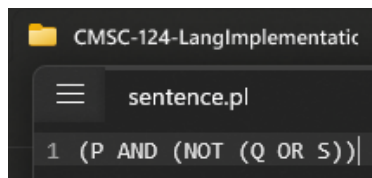
Truth Table of: NOT (P AND (Q OR S))
  P      Q      S      NOT (P AND (Q OR S))
  |      |      |      |
  True   True   True   False
  True   True   False  False
  True   False  True   False
  True   False  False  True
  False  True   True   True
  False  True   False  True
  False  False  True   True
  False  False  False  True

Press any key to continue . . .
```

Press any key to exit the window.

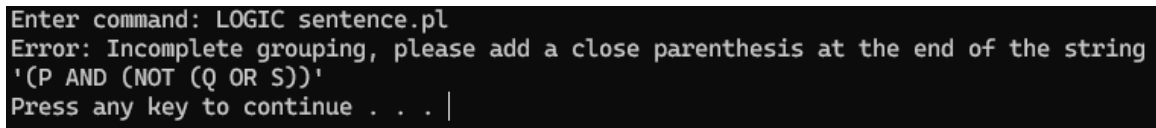
2.5 Editing sentence.pl

You may try editing sentence.pl using a text editor and see which combinations work with the program. Here is an example, using Notepads as the text editor:



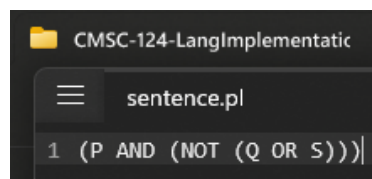
```
CMSC-124-LangImplementatic
sentence.pl
1 (P AND (NOT (Q OR S))|
```

In this example, notice that there is a missing closing parenthesis. Once you run the program, you will get an error message:



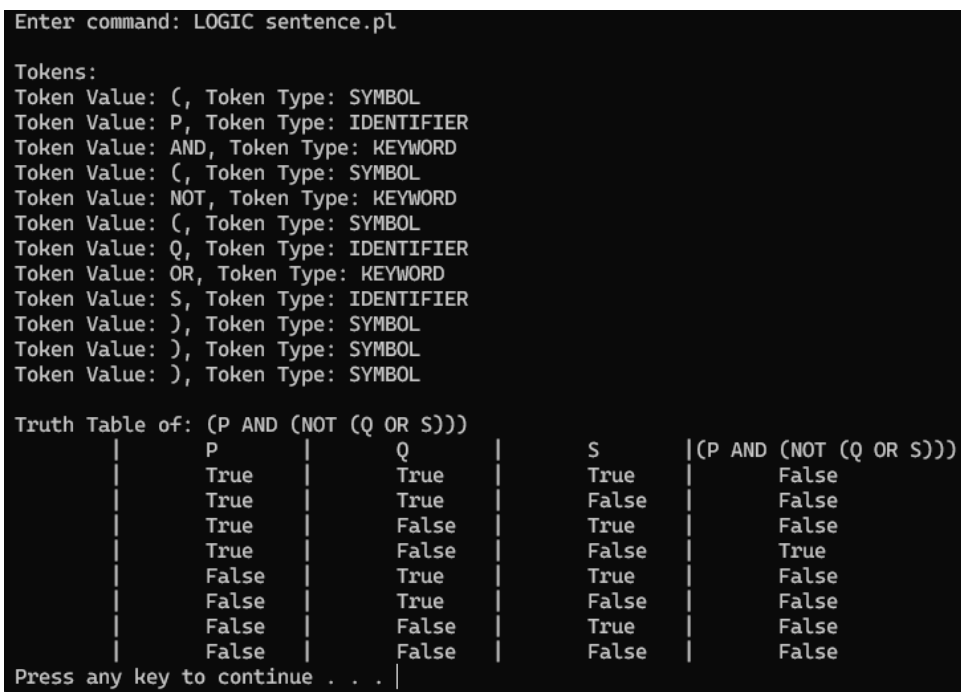
```
Enter command: LOGIC sentence.pl
Error: Incomplete grouping, please add a close parenthesis at the end of the string
'(P AND (NOT (Q OR S))'
Press any key to continue . . . |
```

After correcting the error, this is what the sentence should look like in sentence.pl:



```
CMSC-124-LangImplementatic
sentence.pl
1 (P AND (NOT (Q OR S)))|
```

The program will then generate the corresponding truth table:



```
Enter command: LOGIC sentence.pl

Tokens:
Token Value: (, Token Type: SYMBOL
Token Value: P, Token Type: IDENTIFIER
Token Value: AND, Token Type: KEYWORD
Token Value: (, Token Type: SYMBOL
Token Value: NOT, Token Type: KEYWORD
Token Value: (, Token Type: SYMBOL
Token Value: Q, Token Type: IDENTIFIER
Token Value: OR, Token Type: KEYWORD
Token Value: S, Token Type: IDENTIFIER
Token Value: ), Token Type: SYMBOL
Token Value: ), Token Type: SYMBOL
Token Value: ), Token Type: SYMBOL

Truth Table of: (P AND (NOT (Q OR S)))
  P   Q   S   (P AND (NOT (Q OR S)))
  |   |   |   |
  True True True      False
  True True False     False
  True False True      False
  True False False     True
  False True  True      False
  False True  False     False
  False False True      False
  False False False     False

Press any key to continue . . . |
```

03 Difficulties encountered

During the development of the project, the group experienced minimal internal difficulties.

Technically, the early versions of the project had the processing of tokens lead to erroneous results being printed i.e. wrong truth values in the truth table. This was resolved by debugging the parser.

To be able to collaborate efficiently, the group chose Java as the programming language because of more familiarity with the language as opposed to C++.
