Introduction to language theory and compiling. Project – Part 1

immediate

October 9, 2023

Nom1 Prenom1, Nom2 Prenom2,

Table des matières

1 Makefile Overview

In general, this Makefile provides three main functionalities:

- Building the project
- Testing the project (‘testing‘)
- Cleaning up generated files (‘clean‘)
- The commented-out lines related to Javadoc include documentation generation.

1.1 ‘all‘ target

This target is responsible for the main build process of your project.

- **jflex src/LexicalAnalyzer.flex**: This command uses the JFlex tool to generate the lexer Java code from the ‘LexicalAnalyzer.flex‘ specification. The generated Java file will be named ‘LexicalAnalyzer.java‘ and will be placed in the ‘src/‘ directory.
- **mkdir -p bin**: This command ensures that a ‘bin/‘ directory exists. The ‘-p‘ flag ensures that ‘mkdir‘ doesn’t complain if the directory already exists.
- **jar cfe dist/part1.jar Main -C bin .**: This command packages the compiled bytecode into an executable JAR file named ‘part1.jar‘ in the ‘dist/‘ directory. The ‘Main‘ class is set as the entry point for the JAR, meaning it will be executed when the JAR is run.
- **javadoc -private src/Main.java -d doc/javadoc**: This is a command that generates Javadoc documentation for the ‘Main.java‘ file and places it in the ‘doc/javadoc‘ directory. The ‘-private‘ flag ensures that Javadoc comments from private members are included.

1.2 ’testing‘ target

This target is used to test the program.

- **java -jar dist/part1.jar test/euclid.pmp**: This command runs the previously created ‘part1.jar‘ executable JAR file, passing ‘test/euclid.pmp‘ as an argument. This will tokenize the provided PASCALMAISPRESQUE file and print out the detected tokens and symbol table.

1.3 ‘clean‘ target:

This target is used to clean up generated files, ensuring a fresh build next time.

- **rm -f src/LexicalAnalyzer.java**: This removes the generated ‘Lexical-Analyzer.java‘ lexer file.
- **rm -rf bin/***: This removes all files inside the ‘bin/‘ directory, which contains the compiled bytecode.
- **rm -f dist/part1.jar**: This removes the previously created executable JAR file.
- **rm -rf doc/javadoc/***: This is a commented-out command that would remove all generated Javadoc documentation.

2 Overview of Main.java

The ‘Main‘ class serves as the entry point for the Lexical Analyzer project. Its primary responsibility is to tokenize a given input file based on the provided lexical rules and then print out the sequence of matched lexical units along with a table of recognized variable names.

In summerize, the ‘Main‘ class provides a CLI tool for users to tokenize PASCALMAISPRESQUE files. It reads the file, tokenizes its contents, and prints out the sequence of recognized tokens. Additionally, it maintains and displays a sorted list of all unique variable names encountered in the input file.

2.1 Imports:

- ‘FileReader‘: Used for reading the contents of the input file.
- ‘FileNotFoundException‘ and ‘IOException‘: Exceptions related to file operations.
- ‘TreeMap‘ and ‘Map‘: Used for storing and organizing the recognized variables in a sorted manner.

2.2 Class Declaration:

- The class is publicly accessible and is named ‘Main‘.

2.3 main method:

- It’s the entry point of the program and takes a single argument (the path to the input file).
- There are three exceptions it can throw:
  1. **FileNotFoundException** : This exception is thrown during I/O operations when a file that was expected to be present cannot be found.
  2. **IOException** : This is a more general exception that is thrown for many different I/O-related reasons, such as failed or interrupted I/O operations.
  3. **SecurityException** : This exception is thrown by the security manager to indicate a security violation as accessing a system resource without the necessary permissions.

2.4 Lexical Analysis:

- A new instance of the ‘LexicalAnalyzer‘ (generated by JFlex) is created. This analyzer will be used to tokenize the input.

- A ‘TreeMap‘ named ‘variablesTable‘ is declared to store the recognized variable names. It maps variable names (as strings) to their corresponding ‘Symbol‘ objects. The TreeMap ensures that variables are stored in a sorted (alphabetical) order.

2.5 Tokenization Loop:

- The program enters a loop where it keeps reading tokens from the input file using the ‘nextToken‘ method of the analyzer until it reaches the end of the file (marked by the ‘EOS‘ lexical unit).
- For each token read, it prints out its string representation.
- If the token represents a variable name (‘VARNAME‘), it checks if this variable name is already in the ‘variablesTable‘. If it’s not, the variable name along with its symbol is added to the table. **It will allow to print the first line where the variable appears.**

2.6 Printing Variables:

- After tokenization is completed, the program prints out all the recognized variable names stored in the ‘variablesTable‘ in alphabetical order, along with the line number where each variable was first encountered.