MINISTERUL EDUCAȚIEI, CULTURII ȘI CERCETĂRII AL REPUBLICII MOLDOVA

Universitatea Tehnică a Moldovei

Facultatea Calculatoare, Informatică şi Microelectronică Departamentul Inginerie Software și Automatică

CEBAN VASILE FAF-223

Report

Laboratory work n.1

of Formal Languages and Finite Automata

Checked by:

Dumitru Crețu, *university assistant*

DISA, FCIM, UTM

Chișinău - 2024

1. Regular Grammars Task.
2. Finite Automata Task.
3. Lexer Scanner Task.

1) Regular Grammars Task:

The first task involves implementing a method for generating valid strings from a regular grammar. In this case, the generateString method in the Grammar class accomplishes this task. Let's break down the implementation:

```
public String generateString() {
    return generateStringHelper(startSymbol);
}

private String generateStringHelper(String symbol) {
    if (VT.contains(symbol)) {
        return symbol;
    }

    List<String> productions = P.get(symbol);
  String chosenProduction = productions.get(new Random().nextInt(productions.size()));

    StringBuilder result = new StringBuilder();
    for (char c : chosenProduction.toCharArray()) {
        if (Character.isLetter(c)) {
            result.append(generateStringHelper(String.valueOf(c)));
        } else {
            result.append(c);
        }
    }

    return result.toString();
}
```

Explanation:

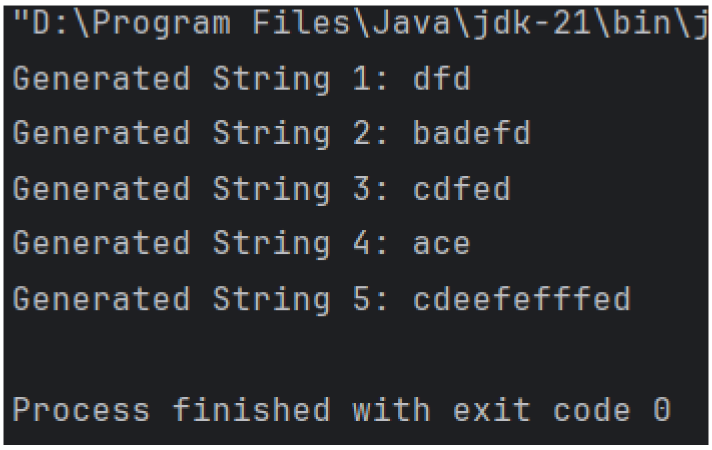
- The generateString method serves as a public entry point for generating strings. It initiates the process by calling the helper method generateStringHelper with the start symbol.
- The generateStringHelper method recursively expands the given symbol based on the production rules until terminal symbols (VT) are reached.
- If the symbol is a terminal symbol (VT), it is simply returned.

- If the symbol is a non-terminal symbol (VN), a random production rule is chosen, and the symbols in that rule are recursively expanded using the generateStringHelper method.
- The result is a string generated following the grammar's production rules.

In the provided Main class, the generateString method is called five times to generate and print five strings:

```
for (int i = 0; i < 5; i++) {
    String generatedString = grammar.generateString();
    System.out.println("Generated String " + (i + 1) + ": " + generatedString);
}
```

Result example:



```
"D:\Program Files\Java\jdk-21\bin\j
Generated String 1: dfd
Generated String 2: badefd
Generated String 3: cdfed
Generated String 4: ace
Generated String 5: cdeefefffed

Process finished with exit code 0
```

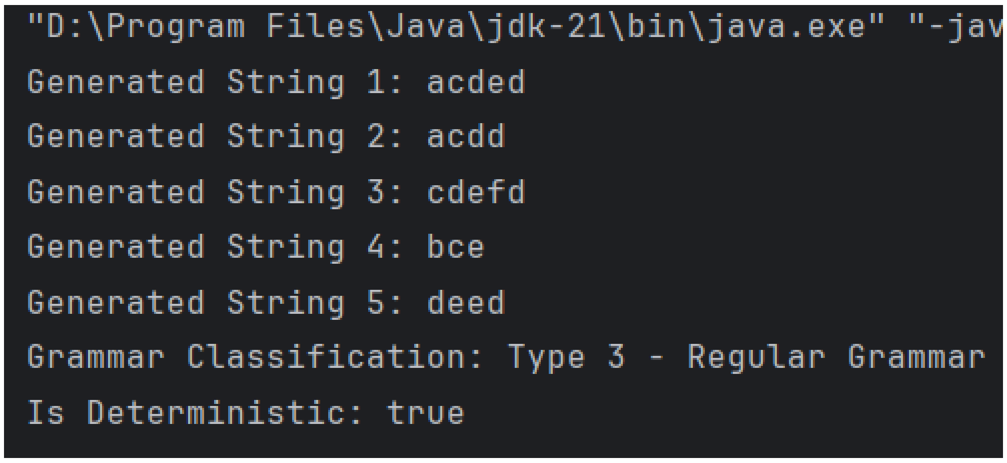
Figure 1. Result at first task(Regular Grammars)

2) Finite Automata Task:

The second task involves working with a finite automaton derived from a regular grammar.

- Type 0 (Unrestricted Grammar): There are no restrictions, meaning any grammar is allowed.
- Type 1 (Context-Sensitive Grammar): Rules have the form $\alpha \rightarrow \beta$ where length(α) <= length(β), and there's at least one rule where length(α) < length(β).
- Type 2 (Context-Free Grammar): Rules have the form $A \rightarrow \gamma$, where A is a nonterminal and γ is a string of terminals and/or nonterminals.
- Type 3 (Regular Grammar): Rules have the form $A \rightarrow aB$ or $A \rightarrow a$, where A and B are nonterminals, and a is a terminal.

Result:



```
"D:\Program Files\Java\jdk-21\bin\java.exe" "-jav
Generated String 1: acded
Generated String 2: acdd
Generated String 3: cdefd
Generated String 4: bce
Generated String 5: deed
Grammar Classification: Type 3 - Regular Grammar
Is Deterministic: true
```

Figure 2. Result at first task(Finite Automata)

3) Lexer Scanner Task:

```
public Lexer(String input) {
   this.tokens = new ArrayList<>();
   tokenize(input);
}
```

The constructor initializes the tokens list and calls the tokenize method to process the input string.

```
private void tokenize(String input) {
      String regex = "\\s*(\\+|\\-|\\*|\\/|\\(|\\)|\\d+|\\w+)\\s*";
      Pattern pattern = Pattern.compile(regex);
      Matcher matcher = pattern.matcher(input);

      while (matcher.find()) {
         String tokenValue = matcher.group(1);
         TokenType tokenType = determineTokenType(tokenValue);
         tokens.add(new Token(tokenType, tokenValue));
      }
   }
```

Uses regular expressions to tokenize the input string and determine the type of each token. The determineTokenType method is called to assign the appropriate TokenType to each token.

```
public enum TokenType {
  NUMBER,
  IDENTIFIER,
  PLUS,
  MINUS,
  MULTIPLY,
  DIVIDE,
  OPEN_PAREN,
  CLOSE_PAREN,
  UNKNOWN
}
```

Determines the type of a token based on its value. Handles cases for numbers, identifiers, and specific operators.

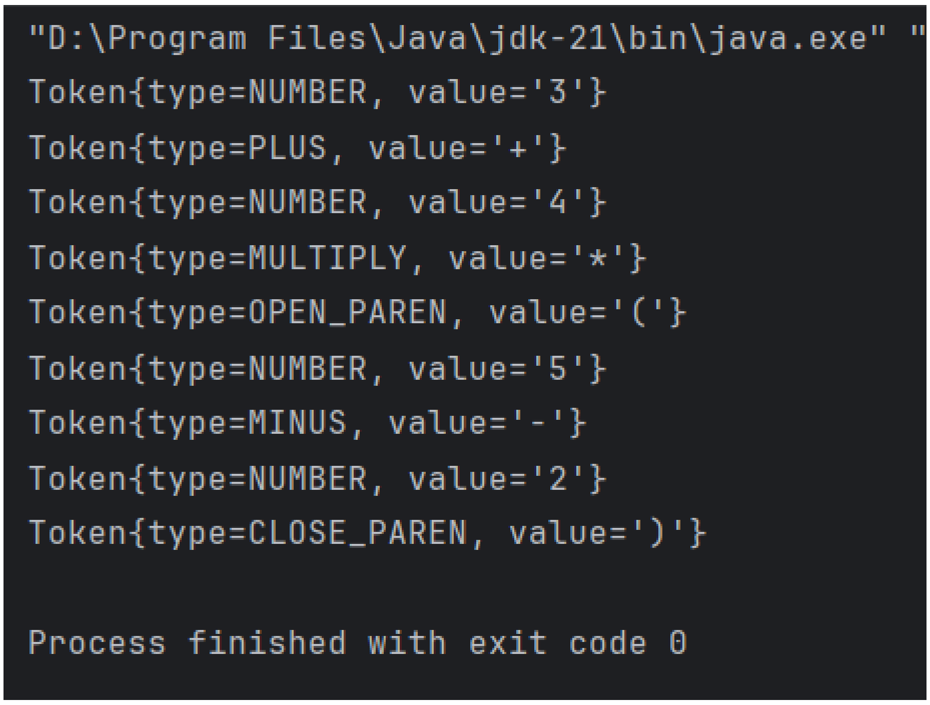
```
public Token(TokenType type, String value) {
    this.type = type;
    this.value = value;
}

public TokenType getType() {
    return type;
}

public String getValue() {
    return value;
}
```

Constructor sets the type and value for a token. Getters to retrieve the token type and value.

Result:



```
"D:\Program Files\Java\jdk-21\bin\java.exe" "
Token{type=NUMBER, value='3'}
Token{type=PLUS, value='+'}
Token{type=NUMBER, value='4'}
Token{type=MULTIPLY, value='*'}
Token{type=OPEN_PAREN, value='('}
Token{type=NUMBER, value='5'}
Token{type=MINUS, value='-'}
Token{type=NUMBER, value='2'}
Token{type=CLOSE_PAREN, value=')'}

Process finished with exit code 0
```

Figure 3. Result at third task(Lexer Scanner)