



**SCHOOL OF COMPUTER SCIENCE**  
**UNIVERSITI SAINS MALAYSIA**

**CPT316 – PROGRAMMING LANGUAGE**  
**IMPLEMENTATION AND PARADIGMS**

Semester 1, Academic Session 2023/2024

**ASSIGNMENT 1**

Lecturer: Dr Nibras Abdullah Ahmed Faqera

Group: 8

No.	Name	Matrics Number
1.	Abdul Qayyum bin Anuar	158085
2.	Michelle Ling Mya Shuet	157926
3.	Huda Nabilah binti Zulkarnain	157634
4.	Fatin Aqilah binti Zukri	155330
5.	Sofea binti Taufik	159072

## **Contents**

Work Distribution .....	3
Introduction.....	3
Lexical Analysis.....	4
Syntax Analysis.....	10
Flow of Program .....	17
Test Cases.....	19
References.....	24

## Work Distribution

Member	Task
Abdul Qayyum bin Anuar	Syntax Analysis
Michelle Ling Mya Shuet	Syntax Analysis
Huda Nabilah binti Zulkarnain	Lexical Analysis
Fatin Aqilah binti Zukri	Lexical Analysis
Sofea binti Taufik	Documentation

## Introduction

This report details on the documentation of a program we have created to perform a lexical and syntax analysis of a simple programming language based on several defined rules. We have chosen Java as the language to perform lexical and syntax analysis. There are several reasons we chose Java, such as portability, libraries supported, and platform independence.

Lexical and syntax analysis are crucial steps in text analysis. The program is designed to recognize different language constructs, identify tokens, and enforce specific rules to ensure code validity. In the lexical analysis, the program will identify the root words in sentences and group it into several types of tokens, namely: *KEYWORD*, *CONSTANT*, *IDENTIFIER*, *LITERAL*, *SYMBOL*, *OPERATOR*, and *SEPARATOR*. Meanwhile, in the syntax analysis the program validates the input's sentence structure and the grammatical rules.

To use the program, simply run the program and input the source code when prompted. The program will report any rules or violations to the programming language in the input if any exists. Else, the program will produce the output list of tokens and the Abstract Syntax Tree.

This report will extensively cover the program's design and workings in terms of lexical and syntax analysis, explain the general flow of the program and demonstrate some test cases that showcases the program's validation of the input string at work.

## Lexical Analysis

In our program, the tokens are recognized based on the defined patterns for each token type. Each token will be stored as an instance of a class once it has been recognised.

```
public static class Token {  
    public final Type t;  
    public final String c;  
    public Token(Type t, String c) {  
        this.t = t;  
        this.c = c;  
    }  
    public String toString() {  
        return t.toString() + "<" + c + ">";  
    }  
}
```

Class:	Token
Use:	The Token class includes variables of data types Type (t) and a String (c). The constructor creates a new Token object with the specified Type and String. The toString() method returns a string representation of the Token object.

```
public static List<Token> lex(String input) {  
    // Define regex patterns for different language constructs  
    String localKeywordPattern = "\\b(return)\\b";  
    String localConstantPattern = "\\b\\d+\\b";  
    String localIdentifierPattern = "\\b(?!return\\b) [a-zA-Z]\\w*\\b";  
    String localLiteralPattern = "\"[^\"]*\"";  
    String localSymbolPattern = "[#@]";  
    String localOperatorPattern = "\\+|-|\\*|/|%|==|!=|<|>|<=|>=|=";  
    String localSeparatorPattern = "\\(|\\)|\\{|\\}|\\;";  
    String localIllegalCharacterPattern = "[^\\s]";  
}
```

Class:	lex
Use:	<p>It takes in a String input and return a List&lt;Token&gt; object. The input will then be categorised as tokens based on the matching patterns.</p> <ul style="list-style-type: none"><li>localKeywordPattern: Matches keywords like “return”</li></ul>

	<ul style="list-style-type: none"> <li>• <code>localConstantPattern</code>: Matches numeric constants like “123”</li> <li>• <code>localIdentifierPattern</code>: Matches identifiers starting with a letter, excluding “return”</li> <li>• <code>localLiteralPattern</code>: Matches string literals enclosed in double quotes</li> <li>• <code>localSymbolPattern</code>: Matches symbols like “#” or “@”</li> <li>• <code>localOperatorPattern</code>: Matches arithmetic and comparison operators</li> <li>• <code>localSeparatorPattern</code>: Matches punctuation like parentheses, curly braces, or semicolons</li> <li>• <code>LocalIllegalCharacterPattern</code>: Identify characters that are not listed</li> </ul> <p>Pattern and Matcher classes from a Java package – <code>java.util.regex</code> is used to identify and extract tokens. It will then add the recognized tokens into a list.</p>
--	--

```
String combinedPattern = String.format(
    "(%s)|(%s)|(%s)|(%s)|(%s)|(%s)|(%s)|(%s)",
    localKeywordPattern,
    localConstantPattern, localIdentifierPattern,
    localLiteralPattern, localSymbolPattern, localOperatorPattern,
    localSeparatorPattern, localIllegalCharacterPattern
);
```

```
Pattern combinedPatternCompiled = Pattern.compile(combinedPattern);
Matcher matcher = combinedPatternCompiled.matcher(input);
```

Use:	<p>The above part of the program is used to combine the varying regular expression into a single pattern. Each regular expression patterns are separated by <code>()</code> operators and are used to match and distinct patterns in input specified by the regular expressions. The <code>String.format()</code> method is used to create a formatted string that contains the individual regular expressions, separated by OR operators <code>()</code>. This creates a single regular expression that can match any of the patterns defined by the individual regular expressions. The <code>Pattern.compile()</code> method is used to compile the combined regular expression into a <code>Pattern</code> object. The <code>Matcher</code> object is then created using the <code>pattern.matcher(input)</code> method. The <code>matcher</code> object can be used to find matches of the regular expression in the input string.</p>
------	---

```
// List to store tokens
List<Token> tokens = new ArrayList<>();
String lastTokenType = null;
boolean semicolonEncountered = false;

// Loop through the input and identify tokens
while (matcher.find()) {
```

```

String matchedGroup = matcher.group();
Type type = null;

// Determine the type of the matched group based on the patterns
if (matchedGroup.matches(localKeywordPattern)) {
    type = Type.KEYWORD;
} else if (matchedGroup.matches(localConstantPattern)) {
    type = Type.CONSTANT;
} else if (matchedGroup.matches(localIdentifierPattern)) {
    type = Type.IDENTIFIER;
} else if (matchedGroup.matches(localLiteralPattern)) {
    type = Type.LITERAL;
} else if (matchedGroup.matches(localSymbolPattern)) {
    type = Type.SYMBOL;
} else if (matchedGroup.matches(localOperatorPattern)) {
    type = Type.OPERATOR;
} else if (matchedGroup.matches(localSeparatorPattern)) {
    type = Type.SEPARATOR;
} else if (matchedGroup.matches(localIllegalCharacterPattern)) {
    checkIllegalCharacter(scanner, matchedGroup);
    continue; // Skip the rest of the loop for illegal characters
}

// Check for rule violations
checkOperator(type, matcher, input);
checkConsecutiveTokens(type, lastTokenType);
checkLiteralsAndConstants(type, lastTokenType, matchedGroup);

// Add the identified token to the list
tokens.add(new Token(type, matchedGroup));
lastTokenType = type.toString();

if (matchedGroup.equals(";")) {
    semicolonEncountered = true;
}
}

// Check for additional rule violations
checkSemicolon(input, semicolonEncountered);

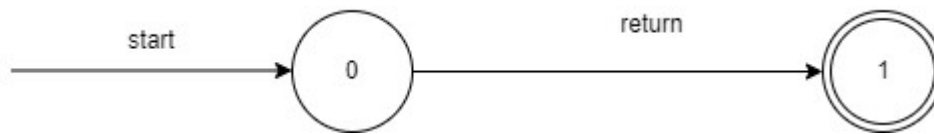
// Return the list of tokens
return tokens;

```

Use:	<p>The program will iteratively check the input for any matching pattern of tokens and assign the matching part of the input string as the matched token type. In each iteration, several functions (checkIllegalCharacter, checkOperator, checkConsecutiveTokens, checkLiteralsandConstant) will be called to check the correctness of the input syntax. The program will also check for the presence of semicolons in the string and will assign true to the sentinel variable semicolonEncountered whenever a semicolon is encountered. Once the loop ends, checkSemicolon will be called to check for syntax validity.</p>
------	--

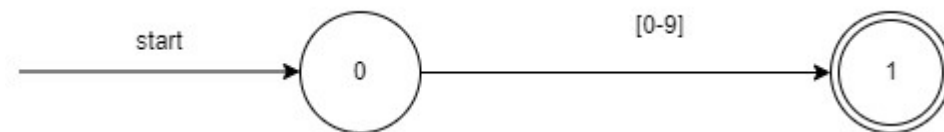
## REGULAR EXPRESSIONS & FINITE AUTOMATA

i) Keyword Regex: return



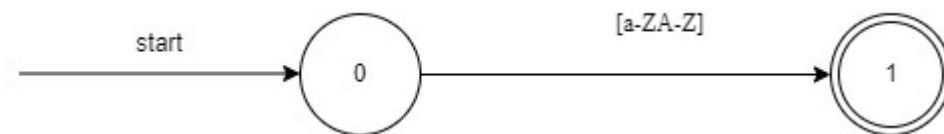
*Finite Automata for Keyword Regular Expression*

ii) Constant Regex: [0-9]



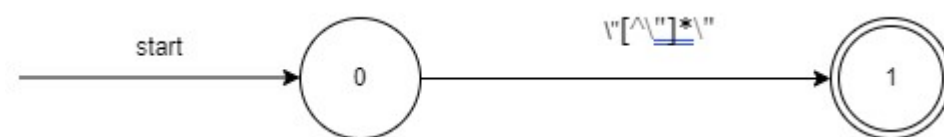
*Finite Automata for Constant Regular Expression*

iii) Identifier Regex: [a-zA-Z]



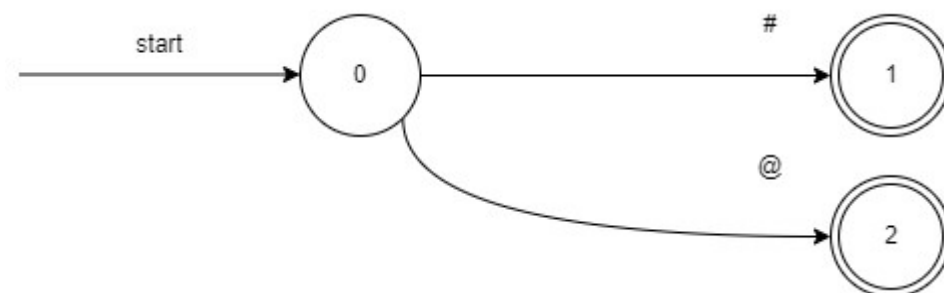
*Finite Automata for Identifier Regular Expression*

iv) Literal Regex: \"[^\"]\*\"



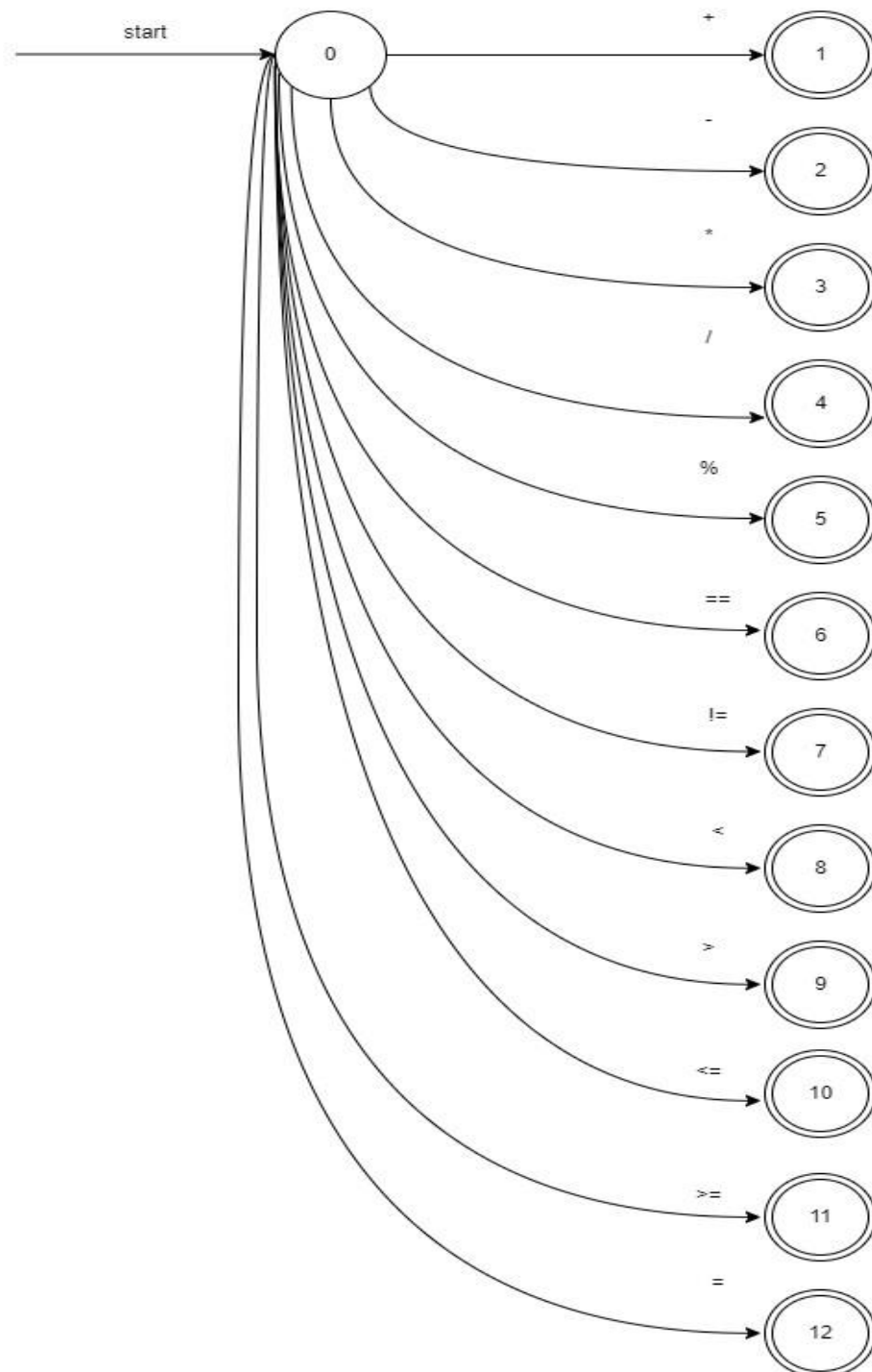
*Finite Automata for Literal Regular Expression*

v) Symbol Regex: #|@



*Finite Automata for Symbol Regular Expression*

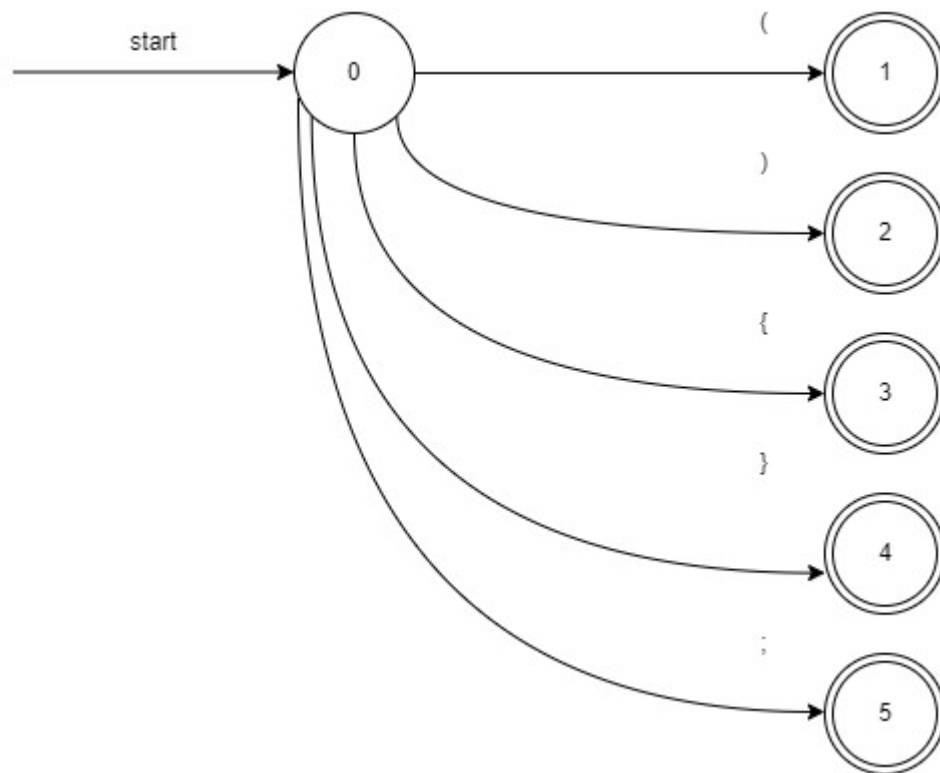
vi) Operator Regex: +|-|\*|%|==|!=|<|>|<=|>=|



*Finite Automata for Operator Regular Expression*



vii) Separator Regex:  $()|\{\} ;$



*Finite Automata for Operator Separator Expression*

## Syntax Analysis

The syntax analysis' purpose is to make sure the string obtained from the input is syntactically correct based on the defined grammatical rules for the programming language. There are several methods implemented in the program, each with a rule that the method checks the input's abundance to. This part of the analysis is also used to parse the tokenised input into an abstract syntax tree to further visualise the sentence structure of the input.

```
//Rule 1: Check if the code starts and ends with curly brackets
private static boolean checkCurlyBracket(String code) {
    code = code.trim();

    if (!code.startsWith("{") || !code.endsWith("}")) {
        System.out.println("\nRule 1 Violation: Source code must start with
{' and end with '}'");
        return false;
    }

    return true;
}
```

Rule 1:	To check if the code starts and ends with curly brackets
Use:	It will trim the input code to see the first and the last character received. This is to check whether it starts with '{' and ends with '}'. If not, it prints a violation message and returns false.

```
// Rule 2: Check for illegal characters in the source code
private static void checkIllegalCharacter(Scanner scanner, String
character) {
    throw new RuleViolationException("\nRule 2 Violation: Source code
cannot have illegal character: " + character);
}
```

Rule 2:	To check for illegal characters in the source code
Use:	Using localIllegalCharacterPattern, it will identify character received that are not listed in any other pattern. Then, it will throw a RuleViolationException with a message indicating a violation if an illegal character is encountered.

```
// Rule 3: Check if the operator is used correctly between two identifiers
private static void checkOperator(Type type, Matcher matcher, String input)
{
    if (type == Type.OPERATOR) {
        int nextTokenIndex = matcher.end();
        while (nextTokenIndex < input.length() &&
Character.isWhitespace(input.charAt(nextTokenIndex))) {
            nextTokenIndex++;
        }

        if (nextTokenIndex < input.length()) {
```

```

        String nextToken = input.substring(nextTokenIndex,
nextTokenIndex + 1);
        Type nextTokenType = null;

        if (nextToken.matches("\\b\\d+\\b")) {
            nextTokenType = Type.CONSTANT;
        } else if (nextToken.matches("\\b(?:return\\b)[a-zA-Z]\\w*\\b")) {
            nextTokenType = Type.IDENTIFIER;
        }

        if (nextTokenType == null ||
(!nextTokenType.equals(Type.IDENTIFIER) &&
!nextTokenType.equals(Type.CONSTANT))) {
            throw new RuleViolationException("\nRule 3 violation:
Operator must be used between two identifiers");
        }
    }
}

```

Rule 3:	To check if the operator is used correctly between two identifiers
Use:	It checks if the current token is an operator and verifies that the previous token was either an identifier or a constant. If not, it throws a RuleViolationException. Additionally, it checks the token following the operator, skipping any whitespaces, to ensure that it is also an identifier or a constant. If not, it throws another RuleViolationException.

```

// Rule 4: Check for consecutive tokens of the same type
private static void checkConsecutiveTokens(Type type, String lastTokenType)
{
    if (type.toString().equals(lastTokenType) && type != Type.SEPARATOR) {
        throw new RuleViolationException("\nRule 4 Violation: Two
consecutive tokens of the same type are not allowed");
    }
}

```

Rule 4:	To check for consecutive tokens of the same type
Use:	Consecutive tokens of the same type are not allowed, except for separators. It checks if the current token type is the same as the last token type and if the type is not a separator. If both conditions are true, it throws a RuleViolationException. This rule helps prevent errors and ensures that the code is easy to read and understand.

```
// Rule 5: Check if literals and constants are used in the correct context
private static void checkLiteralsAndConstants(Type type, String
lastTokenType, String matchedGroup) {
    if ((type == Type.LITERAL || type == Type.CONSTANT)
        && !(lastTokenType != null
            && (lastTokenType.equals(Type.OPERATOR.toString())
                || lastTokenType.equals(Type.KEYWORD.toString())
                || lastTokenType.equals(Type.KEYWORD.toString() + "<return>"))
        && !matchedGroup.equals("return"))) {
        throw new RuleViolationException("\nRule 5 Violation:
Literals/Constants can only be used in assignment and return operations");
    }
}
```

Rule 5:	To check if literals and constants are used in the correct context
Use:	Literals and constants can only be used in assignment and return operations. It checks if the current token type is a literal or constant and if the previous token type is not an operator, a keyword, or a keyword followed by "<return>" (indicating a return statement). Additionally, it checks if the current token is not "return" itself. If any of these conditions are false, it throws a RuleViolationException. This rule ensures that literals and constants are used meaningfully and prevent them from being used in inappropriate contexts.

```
// Rule 6: Check if a semicolon is present before the closing curly brace
private static void checkSemicolon(String input, boolean
semicolonEncountered) {
    if (!semicolonEncountered) {
        int lastSemicolonIndex = input.lastIndexOf(';');
        int lastCurlyBraceIndex = input.lastIndexOf('}');

        if (lastSemicolonIndex < lastCurlyBraceIndex) {
            throw new RuleViolationException("\nRule 6 Violation: Semicolon
is required at the end before the closing curly brace '}'");
        }
    }
}
```

Rule 6:	To check if a semicolon is present before the closing curly braces
Use:	It checks if the input string ends with a semicolon before the closing curly brace '}'. If it doesn't, it throws a RuleViolationException indicating that a semicolon is required at that location. This rule helps ensure consistency and readability of the code, making it easier to understand and maintain.

```
// Rule 7: Check if brackets, parentheses, braces, and brackets are in
pairs
private static boolean checkMatchingPairs(String code) {
    Map<Character, Character> bracketPairs = Map.of('(', ')', '{', '}',
['[', ']']);
}
```

```

java.util.Stack<Character> stack = new java.util.Stack<>();

for (char c : code.toCharArray()) {
    if (bracketPairs.containsKey(c)) {
        stack.push(c);
    } else if (bracketPairs.containsValue(c) && (stack.isEmpty() ||
bracketPairs.get(stack.pop()) != c)) {
        System.out.println("\nRule 7 Violation: Curly brackets,
parentheses, braces, and brackets must be in pairs");
        return false;
    }
}

if (!stack.isEmpty()) {
    System.out.println("\nRule 7 Violation: Curly brackets,
parentheses, braces, and brackets must be in pairs");
    return false;
}
return true;
}

```

Rule 7:	To check if brackets, parentheses, braces, and brackets are in pairs
Use:	It tests whether brackets, brackets, brackets and brackets are properly paired and nested using a stack. It loops through each character, identifying opening and closing bracket pairings using a map. When an opening bracket is discovered, it is pushed onto the stack, and when a closing bracket is met, it validates that it matches the top of the stack's matching opening bracket.

After rules have been checked to make sure the input is grammatically correct, the tokenised input will be parsed to form an Abstract Syntax Tree.

```

public Parser(List<Lexer.Token> tokens) {
    this.tokens = tokens;
    this.currentTokenIndex = 0;
}

```

This is a constructor that receives the `List<Lexer.Token>` and stores it in `tokens`. Then it assigns the `tokens` argument into the `tokens` in this constructor. The variable `currentTokenIndex` is initialised to 0 to ensure that the list starts being processed from the first token in the list.

```

public static class ASTNode {
    private String value;
    private String type;
    private List<ASTNode> children;
}

```

This class defines the structure of Abstract Syntax Tree (AST) node. There are 3 variables which are `value` — stores value such as name of a variable, `type` — stores variable declaration

/ expression / statement, and children — stores list of AST child nodes.

```
// Constructor to create an ASTNode with a value, type, and children
public ASTNode(String value, String type, List<ASTNode> children) {
    this.value           = value;
    this.type            = type;
    this.children        = children;
}
```

These then will be used by the constructor ASTNode to create a new AST node.

```
// Methods for retrieving the value, type, and children of the node
public String getValue()      { return value;    }
public String getType()       { return type;     }
public List<ASTNode> getChildren() { return children; }
}
```

get methods allows other class to get the value of these variables.

```
private void consume() {    currentTokenIndex++;    }
```

Increment of currentTokenIndex to move to the next token for analysis.

```
private Lexer.Token getCurrentToken() {
    if (currentTokenIndex < tokens.size()) {
        return tokens.get(currentTokenIndex);
    }
    return null;
}
```

This function returns the current token without moving to the next token. It checks if the currentTokenIndex is within the bounds of the tokens list. If it is, then it returns the token at that index. Otherwise, it returns null, indicating that there are no more tokens to be processed. This method allows other parts of the parser to access the current token without affecting the parser's position in the token stream.

```
public ASTNode parse() {
    // Start the parsing process with the first child
    ASTNode leftNode = parseChild();
    // Parse the entire expression and return the resulting AST
    ASTNode result = parseNode(leftNode);
    return result;
}
```

The parsing starts by calling the parseChild() then it calls parseNode() to continue parsing the rest of the expression while constructing the AST and return it.

```
// Parse a node in the AST
private ASTNode parseNode(ASTNode leftNode) {
    // Get the current token
```

```

Lexer.Token currentToken = getCurrentToken();

// Skip over whitespace tokens
while (currentToken != null && currentToken.c.equals(" ")) {
    consume();
    currentToken = getCurrentToken();
}

```

Current token is retrieved and any present whitespaces is skipped to ensure that the parser focuses on actual tokens that represent the AST structure.

```

// Check if the current token is of a valid type for a node
if (currentToken != null && isValidTokenTypeForNode(currentToken.t)) {
    // Consume the current token and parse the next child
    consume();
    ASTNode factorNode = parseChild();
    // Create a new node with the operator, its type, and the left and
    right children
    ASTNode newNode = new ASTNode(currentToken.c,
currentToken.t.toString(), List.of(leftNode, factorNode));
    // Recursively parse the next node
    return parseNode(newNode);
}

```

If the current token is not null and is valid type for AST node, it will be consumed and parsed using `parseChild()`. Then it will create a new AST node with its operator, type, and the left and right child nodes. `parseNode()` will be recursively called to continue parsing and passing the newly created node as the starting point.

```

// Return the leftNode if no valid current token is found
return leftNode;

```

If no valid token is found, then it will return the original left node to indicate that there are no further parsing possible.

```

private ASTNode parseChild() {
    // Get the current token
    Lexer.Token currentToken = getCurrentToken();

    // Skip over whitespace tokens
    while (currentToken != null && currentToken.c.equals(" ")) {
        consume();
        currentToken = getCurrentToken();
    }
}

```

The current token is retrieved and the whitespaces token will be skipped where they are present.

```

// Check if there is a valid current token for a child node
if (currentToken != null) {
// Consume the current token and create a node with its value and
type
consume();
return new ASTNode(currentToken.c, currentToken.t.toString(), null);
}

```

If the current token is not null, the code will consume it and create a new AST node. This node will be a single child node.

```

// Return an empty node if no valid current token is found
return new ASTNode("", "", null);
}

```

If the token is not found, then it will return an empty AST node. This indicates that there are no further parsing.

```

private boolean isValidTokenTypeForNode(Lexer.Type type) {
return type == Lexer.Type.CONSTANT || type == Lexer.Type.IDENTIFIER
|| type == Lexer.Type.KEYWORD || type == Lexer.Type.LITERAL
|| type == Lexer.Type.SYMBOL || type == Lexer.Type.OPERATOR
|| (type == Lexer.Type.SEPARATOR &&
!getCurrentToken().c.equals("{"));
}

```

In this method, the token provided by lexical analysis will be validated to ensure that they can be used for AST node, then return value true or false.



## Flow of Program

```
public static void main(String[] args) {
    Scanner scanner = new Scanner(System.in);

    while (true) {
        System.out.println("Enter source code (Your code needs to start and
end with curly brackets):");
        String input = scanner.nextLine();
        if (checkCurlyBracket(input) && checkMatchingPairs(input)) {
            try {
                // Lexical analysis
                List<Lexer.Token> tokens = Lexer.lex(input);

                // Display tokens
                System.out.println("\nTokens:");

                for (Lexer.Token t : tokens) {
                    System.out.printf("%-15s%s\n", t.t, t.c);
                }
            }
        }
    }
}
```

1. Once the program starts, the program will prompt the user for an input to enter a source code. The input source code must start and end with curly brackets. The program will check whether this condition is fulfilled and determine whether parentheses, brackets and braces are balanced both with its opening and closing symbols via *checkCurlyBracket* and *checkMatchingPairs* functions. Then, the program will tokenise the input using the *lex* function once the conditions are passed and print out the assigned tokens.
2. Inside the *lex* function, the program will group the input into tokens. The types of tokens are *KEYWORD*, *CONSTANT*, *IDENTIFIER*, *LITERAL*, *SYMBOL*, *OPERATOR*, and *SEPARATOR*. During this stage, *checkIllegalCharacter*, *checkOperator*, *checkConsecutiveTokens*, *checkLiteralsAndConstants* and *checkSemicolon* will be called to check for grammatical rules for the input. If the input does not abide by the rules, an error message will be printed.

```
Parser parser = new Parser(tokens);

try {
    Parser.ASTNode root = parser.parse();
    System.out.println("\nAbstract Syntax Tree (AST):");
    Parser.printAST(root, 0);
} catch (RuntimeException e) {
    System.out.println(e.getMessage());
}

} catch (RuleViolationException e) {
```

```

        System.out.println(e.getMessage());
    }

```

3. tokens obtained from the lexical stage will be passed to be parsed. A new parser object will be created under the Parser class using the new Parser() function;

```

Parser      parser      =      new      Parser(tokens);

```

The tokens will go through parsing and immediately start constructing an AST then printed out.

```

        if (!askForAnotherInput(scanner)) {
            break; // Exit the loop if the user doesn't want to enter
another source code
        }

        } else {
            if (!askForAnotherInput(scanner)) {
                break; // Exit the loop if the user doesn't want to re-
enter the source code
            }
        }
    }

    scanner.close();
}

```

4. Finally, the program will prompt the user for another input through the askForAnotherInput. If the user enters “yes”, the loop will iterate again to let the user input again. Else, the program will exit.

```

private static boolean askForAnotherInput(Scanner scanner) {
    System.out.println("\n-----\nDo you want
to enter another source code? (yes/no)");
    String response = scanner.nextLine().trim().toLowerCase();
    if (!response.equals("yes")) {
        return false;
    } else {
        for (int i = 0; i < 50; i++) {
            System.out.println(); // Print empty lines to "clear" the
console
        }
        return true;
    }
}

```

## Test Cases

### Rule 1: Source code must start with and end with curly brackets

Invalid input	<p> </p> <p>"C:\Users\Sofea Taufik\.jdk\corretto-19.0.2\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA Co</p> <p>Enter source code (Your code needs to start and end with curly brackets):</p> <p><code>x + y = 9</code></p> <p>Rule 1 Violation: Source code must start with '{' and end with '}'</p> <p>-----</p> <p>Do you want to enter another source code? (yes/no)</p>																
Valid input	<p> </p> <p>Enter source code (Your code needs to start and end with curly brackets):</p> <p><code>{meow + 2 = y;}</code></p> <p>-----</p> <p>Tokens:</p> <table border="0"> <tr><td>SEPARATOR</td><td>{</td></tr> <tr><td>IDENTIFIER</td><td>meow</td></tr> <tr><td>OPERATOR</td><td>+</td></tr> <tr><td>CONSTANT</td><td>2</td></tr> <tr><td>OPERATOR</td><td>=</td></tr> <tr><td>IDENTIFIER</td><td>y</td></tr> <tr><td>SEPARATOR</td><td>;</td></tr> <tr><td>SEPARATOR</td><td>}</td></tr> </table> <p>-----</p> <p>Abstract Syntax Tree (AST):</p> <pre> SEPARATOR      }   IDENTIFIER    y     CONSTANT    2       IDENTIFIER meow         SEPARATOR {           OPERATOR +             OPERATOR =               SEPARATOR ;   </pre> <p>-----</p> <p>Do you want to enter another source code? (yes/no)</p>	SEPARATOR	{	IDENTIFIER	meow	OPERATOR	+	CONSTANT	2	OPERATOR	=	IDENTIFIER	y	SEPARATOR	;	SEPARATOR	}
SEPARATOR	{																
IDENTIFIER	meow																
OPERATOR	+																
CONSTANT	2																
OPERATOR	=																
IDENTIFIER	y																
SEPARATOR	;																
SEPARATOR	}																
AST																	

### Rule 2: Source code cannot have illegal character

Invalid input	<p>Enter source code (Your code needs to start and end with curly brackets):</p> <p><code>{Dr. Nibras;}</code></p> <p>Rule 2 Violation: Source code cannot have illegal character: .</p> <p>-----</p> <p>Do you want to enter another source code? (yes/no)</p>
---------------	---

Valid input	<p>Enter source code (Your code needs to start and end with curly brackets):</p> <pre>{CPT316;}</pre> <p>-----</p> <p>Tokens:</p> <pre>SEPARATOR    { IDENTIFIER    CPT316 SEPARATOR    ; SEPARATOR    }</pre> <p>-----</p> <p>Abstract Syntax Tree (AST):</p> <pre>SEPARATOR    }   IDENTIFIER    CPT316     SEPARATOR    {       SEPARATOR    ;</pre> <p>-----</p> <p>Do you want to enter another source code? (yes/no)</p>
AST	

### Rule 3: Operator must be used between two identifiers

Invalid input	<p>Enter source code (Your code needs to start and end with curly brackets):</p> <pre>{x + + y;}</pre> <p>Rule 3 violation: Operator must be used between two identifiers</p> <p>-----</p> <p>Do you want to enter another source code? (yes/no)</p>
Valid input	<p>Enter source code (Your code needs to start and end with curly brackets):</p> <pre>{y + z;}</pre> <p>-----</p> <p>Tokens:</p> <pre>SEPARATOR    { IDENTIFIER    y OPERATOR      + IDENTIFIER    z SEPARATOR    ; SEPARATOR    }</pre> <p>-----</p> <p>Abstract Syntax Tree (AST):</p> <pre>SEPARATOR    }   IDENTIFIER    z     IDENTIFIER    y       SEPARATOR    {         OPERATOR    +         SEPARATOR    ;</pre> <p>-----</p> <p>Do you want to enter another source code? (yes/no)</p>
AST	

### Rule 4: Two consecutive tokens of the same type are not allowed

Invalid input	<p>Enter source code (Your code needs to start and end with curly brackets):</p> <pre>{s s;}</pre> <p>Rule 4 Violation: Two consecutive tokens of the same type are not allowed</p> <p>-----</p> <p>Do you want to enter another source code? (yes/no)</p>
Valid input	<p>Enter source code (Your code needs to start and end with curly brackets):</p> <pre>{ss;}</pre> <p>-----</p> <p>Tokens:</p> <pre>SEPARATOR    { IDENTIFIER    ss SEPARATOR    ; SEPARATOR    }</pre> <p>-----</p> <p>Abstract Syntax Tree (AST):</p> <pre>SEPARATOR    }   IDENTIFIER  ss     SEPARATOR {       SEPARATOR ;</pre> <p>-----</p> <p>Do you want to enter another source code? (yes/no)</p>
AST	<pre>       }      /     ss    /  \   { s ; </pre>

### Rule 5: Literals/Constants can only be used in assignment and return operations


Invalid input	<p>Enter source code (Your code needs to start and end with curly brackets):</p> <pre>{"cat";}</pre> <p>Rule 5 Violation: Literals/Constants can only be used in assignment and return operations</p> <p>-----</p> <p>Do you want to enter another source code? (yes/no)</p>
Valid input	<p>Enter source code (Your code needs to start and end with curly brackets):</p> <pre>{cat;}</pre> <p>-----</p> <p>Tokens:</p> <pre>SEPARATOR    { IDENTIFIER    cat SEPARATOR    ; SEPARATOR    }</pre> <p>-----</p> <p>Abstract Syntax Tree (AST):</p> <pre>SEPARATOR    }   IDENTIFIER  cat     SEPARATOR {       SEPARATOR ;</pre> <p>-----</p> <p>Do you want to enter another source code? (yes/no)</p>

AST	<pre> graph TD     cat[cat] --- L1["{"]     cat --- L2[";"] </pre>
-----	--

Rule 6: Semicolon is required at the end before the closing curly brace '}'

Invalid input	<p>Enter source code (Your code needs to start and end with curly brackets):</p> <pre>{x + y = 9}</pre> <p>Rule 6 Violation: Semicolon is required at the end before the closing curly brace '}'</p> <p>-----</p> <p>Do you want to enter another source code? (yes/no)</p>
Valid input	<p>Enter source code (Your code needs to start and end with curly brackets):</p> <pre>{x + y = 9;}</pre> <p>-----</p> <p>Tokens:</p> <pre> SEPARATOR      { IDENTIFIER      x OPERATOR        + IDENTIFIER      y OPERATOR        = CONSTANT        9 SEPARATOR      ; SEPARATOR      } </pre> <p>-----</p> <p>Abstract Syntax Tree (AST):</p> <pre> SEPARATOR      }   CONSTANT      9     IDENTIFIER   y       IDENTIFIER  x         SEPARATOR {           OPERATOR +           OPERATOR =           SEPARATOR ; </pre> <p>-----</p> <p>Do you want to enter another source code? (yes/no)</p>
AST	<pre> graph TD     y[y] --- L1["{"]     y --- L2["+"]     y --- L3["="]     y --- L4["9"]     x[x] --- L5["{"]     x --- L6["+"] </pre>

## Rule 7: Curly brackets, parentheses, braces, and brackets must be in pairs

Invalid input	<p>Enter source code (Your code needs to start and end with curly brackets):</p> <pre>{(Free Palestine;}</pre> <p>Rule 7 Violation: Curly brackets, parentheses, braces, and brackets must be in pairs</p> <p>-----</p> <p>Do you want to enter another source code? (yes/no)</p>
Valid input	<p>Enter source code (Your code needs to start and end with curly brackets):</p> <pre>{(Finger);}</pre> <p>-----</p> <p>Tokens:</p> <pre>SEPARATOR { SEPARATOR ( IDENTIFIER Finger SEPARATOR ) SEPARATOR ; SEPARATOR }</pre> <p>-----</p> <p>Abstract Syntax Tree (AST):</p> <pre>SEPARATOR }   SEPARATOR )     SEPARATOR (       SEPARATOR {         IDENTIFIER Finger       SEPARATOR ;     SEPARATOR }</pre> <p>-----</p> <p>Do you want to enter another source code? (yes/no)</p>
AST	 <pre>       }      /     )    /  \   {    ;  /  \ {    Finger </pre>

## References

- Roulo, M. (1997, May 1). *Java's three types of portability*. InfoWorld.  
<https://www.infoworld.com/article/2076944/java-s-three-types-of-portability.html>
- GeeksforGeeks. (2023, September 27). *Introduction of lexical analysis*.  
<https://www.geeksforgeeks.org/introduction-of-lexical-analysis/>
- GeeksforGeeks. (2021, August 12). *Abstract Syntax Tree AST in Java*.  
<https://www.geeksforgeeks.org/abstract-syntax-tree-ast-in-java/>
- GeeksforGeeks. (2022, February 16). *Matcher class in Java*.  
<https://www.geeksforgeeks.org/matcher-class-in-java/>
- GeeksforGeeks. (2023, April 19). *Introduction to Syntax analysis in Compiler Design*.  
<https://www.geeksforgeeks.org/introduction-to-syntax-analysis-in-compiler-design/>
- Hyer, A. (2023, March 24). *Java: Creating a Lexical Analyzer*. CopyProgramming.  
<https://copyprogramming.com/howto/making-a-lexical-analyzer>