C

Se

439

Project Design of compilers(lexer part)

Spring

202

4

**Team Members:**

**Yousef mohamed zaki 21p0079**

**Eslam Ashraf fathy 21p0403**

**Mostafa hassan mohamed 21p0349**

**Adel mohamed Adel El-said 21p0113**

**Ahmed Mohamed Salah 2100669**



[Lexer Analyzer 2](#_Toc162379374)

[Over view about Lexer: 2](#_Toc162379375)

[Lexer Responsibility: 3](#_Toc162379376)

[Input and Output 3](#_Toc162379377)

[Regular Languages: 3](#_Toc162379378)

[Tokens 3](#_Toc162379379)

[Tokens : 3](#_Toc162379380)

[Tokens overview: 3](#_Toc162379381)

[Tokens types: 4](#_Toc162379382)

[1. Keywords 4](#_Toc162379383)

[2. Literals 4](#_Toc162379384)

[3. Identifiers 4](#_Toc162379385)

[4. Symbols 4](#_Toc162379386)

[5. Operators 4](#_Toc162379387)

[Lexer Implementation : 4](#_Toc162379388)

[The design of the lexer: 4](#_Toc162379389)

[Lexer Class: 4](#_Toc162379390)

[The code: 5](#_Toc162379391)

[Code Description: 9](#_Toc162379392)

[Token Class: 10](#_Toc162379393)

[The Code : 10](#_Toc162379394)

[code description: 11](#_Toc162379395)

[TokenType Class: 11](#_Toc162379396)

[The code : 11](#_Toc162379397)

[The code description : 12](#_Toc162379398)

[Test Cases : 12](#_Toc162379399)

# 

# Lexer Analyzer

## Overview about Lexer:

A lexical analyzer is the first phase of a compiler in programming languages. It takes a modified source code as input and outputs a series of tokens, [A Lexer also known as a tokenizer or scanner, is a program that transforms an input stream of characters into a sequence of tokens](https://dev.to/cad97/what-is-a-lexer-anyway-4kdo). [These tokens are the smallest individual units in terms of programming.](https://www.geeksforgeeks.org/c-lexical-analyser-lexer/)

## Lexer Responsibility:

[Input and Output: A lexer reads an input character or byte stream (i.e., characters, binary data, etc.), divides it into tokens using patterns specified in a grammar file or in the code, and generates a token stream as output](https://www.geeksforgeeks.org/lexical-analysis-and-syntax-analysis/)

Regular Languages: Formally, a lexer recognizes some set of Regular languages. A “regular” language is one that can be parsed without any extra state in a single non-backtracking pass. [This makes it very efficient: you only have to look at one byte at a time to make decisions](https://dev.to/cad97/what-is-a-lexer-anyway-4kdo).

[Tokens: Tokens include Keyword, Identifier, Operator, Literal, and Punctuation3](https://stackoverflow.com/questions/11376089/what-is-the-purpose-of-a-lexer). For example, the following are some lexical tokens: Keywords: int, String, long, etc. Identifier: x, y, i, j, num etc. Operators: +,-,\*,/ etc. Literals: 108, 9, 12, 15 etc. Punctuations: , ; . .

## Tokens :

### Tokens overview:

The lexer plays a crucial role in the process of compilation. It simplifies the parsing stage by breaking down the code into small tokens, which are then fed to the parser for further processing. The lexer’s responsibilities are fundamental to the successful operation of a compiler when the lexer identify the tokens correctly and in details.

### Tokens types:

#### 1. Keywords

[Keywords are predefined, reserved words used in programming that have special meanings to the compiler1](https://www.geeksforgeeks.org/keywords-in-c/). These are part of the syntax and cannot be used as identifiers in the program. For example, int, if, while, for, switch, return, etc. are keywords in many programming languages.

#### 2. Literals

[In computer science, a literal is a textual representation (notation) of a value as it is written in source code6](https://en.wikipedia.org/wiki/Literal_%28computer_programming%29). Almost all programming languages have notations for atomic values such as integers, floating-point numbers, and strings, and usually for booleans and characters. For example, 10, 3.14, "Hello, World!", true, false are all literals.

#### 3. Identifiers

[Identifiers are unique names that are assigned to variables, structs, functions, and other entities](https://www.geeksforgeeks.org/c-identifiers/). They are used to uniquely identify the entity within the program. For example, x, totalSum, printMessage, EmployeeRecord are identifiers.

#### 4. Symbols

[In programming, symbols are primitive data types whose instances have a unique human-readable form](https://en.wikipedia.org/wiki/Symbol_%28programming%29). In some programming languages, they are called atoms. Uniqueness is enforced by holding them in a symbol table. Symbols can be used as identifiers.

#### 5. Operators

[Operators in programming are symbols or keywords that represent computations or actions performed on operands14](https://www.geeksforgeeks.org/operators-programming/)[15](https://en.wikipedia.org/wiki/Operator_%28computer_programming%29)[16](https://www.techtarget.com/whatis/definition/operator)[17](https://www.techopedia.com/definition/3485/operator-programming). They play a crucial role in performing various tasks, such as arithmetic calculations, logical comparisons, bitwise operations, etc. For example, +, -, \*, /, ==, !=, &&, ||, ++, -- are operators.

## Lexer Implementation :

### The design of the lexer:

Lexer Class: The Lexer class is responsible for converting a sequence of characters into a sequence of tokens. Tokens are the smallest units of meaning that a program can understand. This is the main class that controls the process of converting a sequence of characters into a sequence of tokens. It reads characters from the input string, identifies the tokens, and adds them to a list of tokens. It also handles errors if it encounters an unexpected sequence of characters.

#### The code:

package com.example.c\_compiler;

import java.util.\*;

public class Lexer {

    private String input;

    private int currentPosition;

    private static final String[] KEYWORDS = {"auto", "break", "case", "char", "const", "continue", "default", "do", "double", "else", "enum", "extern", "float", "for", "goto", "if", "int", "long", "register", "return", "short", "signed", "sizeof", "static", "struct", "switch", "typedef", "union", "unsigned", "void", "volatile","while"};

    List<Token> tokens;

    protected SymbolTable symbolTable;

    public Lexer(String input, SymbolTable symbolTable) {

        this.input=input;

        this.currentPosition = 0;

        tokens = new ArrayList<>();

        this.symbolTable = symbolTable;

    }

    public void tokenize() {

        StringBuilder buffer = new StringBuilder();

        String op;

        input = removeComments();

        while (currentPosition < input.length()) {

            char currentChar = input.charAt(currentPosition);

            if(Character.toString(currentChar).matches("[+\\-\*/%&|<>!=^~?:(),;\\[\\]{}#\\s]") ) {

                if(buffer.length() > 0) {

                    addToken(buffer);

                }

                op = Character.toString(currentChar);

                if(op.matches("[-+\*/%&|<>^!~=]")) {

                    currentPosition++;

                    if( op.equals("=") && Character.toString(input.charAt(currentPosition)).matches("[-+\*/%&|<>^!~]") ){

                        tokens.add(new Token(TokenType.ASSIGN,"="));

                        continue;

                    }else if( currentPosition < input.length() && Character.toString(input.charAt(currentPosition)).matches("[-+\*/%&|<>^!~=]") ) {

                        op += Character.toString(input.charAt(currentPosition));

                    }else {

                        currentPosition--;

                    }

                    tokens.add(new Token( recognizeOperator(op) ,op));

                }else if ( !op.equals("\s") && !op.equals("\n") ){

                    tokens.add(new Token(TokenType.SYMBOL,op));

                    if ( op.equals("{") ){

                        symbolTable.startScope();

                    } else if ( op.equals("}") ) {

                        symbolTable.endScope();

                    }

                }

                op = "";

                buffer.delete(0, buffer.length());

            }else{

                StringBuilder b = new StringBuilder();

                if ( input.charAt(currentPosition) == '"' ){

                    currentPosition++;

                    while ( input.charAt(currentPosition) != '"' ){

                        b.append(input.charAt(currentPosition));

                        currentPosition++;

                    }

                    tokens.add(new Token(TokenType.STRING,b.toString()));

                }else {

                    buffer.append(input.charAt(currentPosition));

                }

            }

            currentPosition++;

        }

        if(buffer.length() > 0) {

            addToken(buffer);

        }

    }

    public void addToken(StringBuilder buffer) {

        String sbuffer = buffer.toString();

        if(is\_keyword(sbuffer)) {

            tokens.add(new Token(TokenType.KEYWORD, sbuffer));

        } else if(sbuffer.matches("[a-zA-Z\_$][a-zA-Z0-9\_$]\*")) {

            Token id = new Token(TokenType.IDENTIFIER, sbuffer);

            tokens.add(id);

            if( tokens.size() > 0 ) {

                Token pre\_token = tokens.get(tokens.size() - 1);

                if ( pre\_token.getType() == TokenType.KEYWORD ){

                    id.setId\_type(pre\_token.getValue());

                    symbolTable.addSymbol(sbuffer,id);

                }

            }

        } else if(sbuffer.matches("\\\".\*?\\\"")) {

            tokens.add(new Token(TokenType.STRING, sbuffer));

        }else if( sbuffer.matches("'(\\\\.|[^'\\\\])\*'")){

            tokens.add(new Token(TokenType.Character,sbuffer));

        }else if( sbuffer.matches("[-+]?[0-9]\*[.]?[0-9]+([eE][-+]?[0-9]+)?") || sbuffer.matches("[-+]?[1-9][0-9]\*") || sbuffer.matches("[-+]?0[bB][01]+") || sbuffer.matches("[-+]?0[xX][0-9a-fA-F]+") || sbuffer.matches("[-+]?0[0-7]\*")){

            if( tokens.size() > 0 ) {

                Token t1 = tokens.get(tokens.size() - 1);

                Token t2 = tokens.get(tokens.size() - 2);

                if ( (t1.getType() == TokenType.SUB || t1.getType() == TokenType.ADD ) && !(t2.getType() == TokenType.IDENTIFIER || t2.getType() == TokenType.INC || t2.getType() == TokenType.DEC)  ) {

                    sbuffer = t1.getValue() + sbuffer;

                    tokens.remove(tokens.size() - 1);

                }

            }

            if( sbuffer.matches("[-+]?[1-9][0-9]\*")){

                tokens.add(new Token(TokenType.DECIMAL,sbuffer));

            }else if( sbuffer.matches("[-+]?0[bB][01]+")){

                tokens.add(new Token(TokenType.BINARY,sbuffer));

            }else if( sbuffer.matches("[-+]?0[xX][0-9a-fA-F]+")){

                tokens.add(new Token(TokenType.HEX,sbuffer));

            }else if( sbuffer.matches("[-+]?0[0-7]\*")){

                tokens.add(new Token(TokenType.OCTAL,sbuffer));

            }else if( sbuffer.matches("[-+]?[0-9]\*[.]?[0-9]+([eE][-+]?[0-9]+)?") ){

                tokens.add(new Token(TokenType.FLOAT,sbuffer));

            }else {

                System.out.println("Syntax Error here :" + sbuffer);

            }

        }else{

                System.out.println("Syntax Error here :" + sbuffer);

            }

        buffer.delete(0, buffer.length());

    }

    public boolean is\_keyword(String str){

        for(String s: KEYWORDS ){

            if ( s.equals(str) ){

                return true;

            }

        }

        return false;

    }

    public TokenType recognizeOperator(String operator) {

        // Mapping operator symbols to their types

        switch (operator) {

            case "+":

                return TokenType.ADD;

            case "-":

                return TokenType.SUB;

            case "\*":

                return TokenType.MUL;

            case "/":

                return TokenType.DIV;

            case "%":

                return TokenType.MOD;

            case "&":

                return TokenType.BIT\_AND;

            case "|":

                return TokenType.BIT\_OR;

            case "^":

                return TokenType.BIT\_XOR;

            case "~":

                return TokenType.BIT\_NOT;

            case "++":

                return TokenType.INC;

            case "--":

                return TokenType.DEC;

            case ">":

                return TokenType.GT;

            case "<":

                return TokenType.LT;

            case ">=":

                return TokenType.GE;

            case "<=":

                return TokenType.LE;

            case "==":

                return TokenType.EQ;

            case "!=":

                return TokenType.NE;

            case "&&":

                return TokenType.AND;

            case "||":

                return TokenType.OR;

            case "!":

                return TokenType.NOT;

            case "<<":

                return TokenType.LEFT\_SHIFT;

            case ">>":

                return TokenType.RIGHT\_SHIFT;

            case "=":

                return TokenType.ASSIGN;

            case "+=":

                return TokenType.ADD\_ASSIGN;

            case "\*=":

                return TokenType.MUL\_ASSIGN;

            case "/=":

                return TokenType.DIV\_ASSIGN;

            case "-=":

                return TokenType.SUB\_ASSIGN;

            case "%=":

                return TokenType.MOD\_ASSIGN;

            default:

                return TokenType.UnknownOP;

        }

    }

    public String removeComments() {

        String pattern = "(//[^\\n]\*)|(/\\\*[^/]\*\\\*/)";

        return input.replaceAll(pattern,"");

    }

    public void set\_ids\_values(Token token){

        for ( Token t : tokens ){

        }

    }

}

#### Code Description:

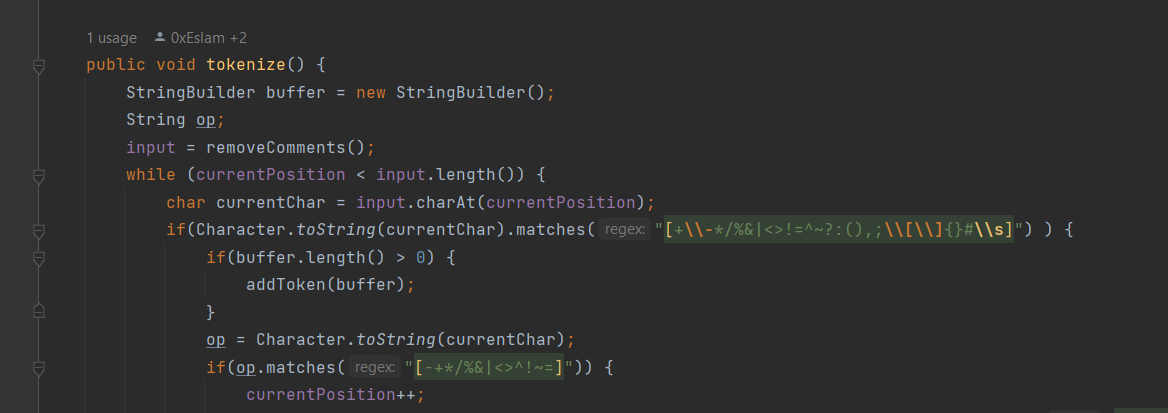
Fields

* input: This is the string that the Lexer will tokenize.
* currentPosition: This is the current position in the input string that the Lexer is examining.
* KEYWORDS: This is a list of all the keywords in C that the Lexer should recognize.
* tokens: This is a list of all the tokens that the Lexer has recognized so far.
* symbolTable: This is a symbol table that the Lexer uses to keep track of identifiers.

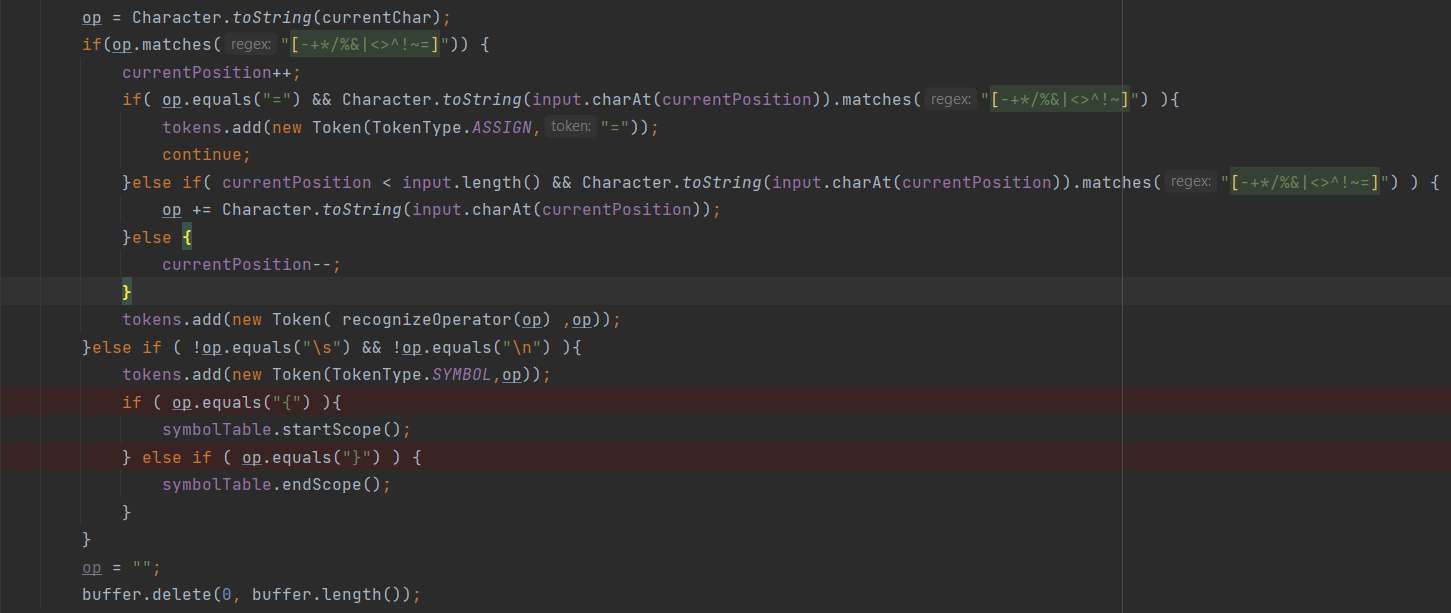
Methods

* Lexer(String input, SymbolTable symbolTable): This is the constructor for the Lexer class. It initializes the input, currentPosition, tokens, and symbolTable.
* tokenize(): This is the main method that performs the lexical analysis. It reads characters from the input string one at a time, identifies the tokens, and adds them to the tokens list.

Process Flow of tokenize():

1. Initialization: The method starts by creating an empty buffer and setting the currentPosition to zero.
2. Reading Characters: The method enters a loop that continues until the end of the input string is reached regard to some delimiters as a regex [+\-\*/%&|<>!=^~?:(),;\[\]{}#\s]
3. Character Categorization: Each character is examined to determine its category:

* If it’s a special character (like operators or punctuation), the buffer is checked by using recognizeOperator() function. If the buffer is not empty, it means a potential token is ready to be created.



* If it’s a regular character (part of an identifier, keyword, or number), it’s appended to the buffer. These tokens detected by the tokenize() function.

1. Token Creation: When a token boundary is detected, the buffer’s content is used to create a new token. The type of the token is determined by pattern matching and keyword checking.
2. Adding Tokens: The new token is added to the tokens list, and the buffer is cleared for the next token.
3. Handling Operators: If the character is an operator, additional logic is applied to check for multi-character operators (like “++” or “==”).
4. Handling Literals: For string literals, a separate buffer (b) is used to capture characters until the closing quote is found.
5. Handling Numbers: Numeric literals are identified using regular expressions to match different number formats (decimal, binary, hexadecimal, etc.).
6. Error Handling: If a sequence of characters does not match any known token pattern, a syntax error is reported.

addToken(StringBuilder buffer): The addToken method is a crucial component of the Lexer class, responsible for creating and adding tokens to the tokens list. This method is called when a sequence of characters in the input string has been identified as a potential token.

The Role of StringBuilder buffer

The buffer parameter is a StringBuilder object that contains the characters that form the token. The buffer is built up in the tokenize method and passed to addToken when a token boundary is detected.

Process Flow of addToken()

1. Conversion to String: The contents of the buffer are converted into a string, referred to as sbuffer, which represents the potential token.
2. Keyword Check: The method first checks if sbuffer is a keyword by calling the is\_keyword method. If it is, a new Token of type KEYWORD is created and added to the tokens list.
3. Identifier and Literal Handling: If sbuffer is not a keyword, the method uses regular expressions to determine if it matches the pattern of an identifier or a literal (string, character, number).
4. Token Creation:

For identifiers, a new Token of type IDENTIFIER is created. If the previous token is a keyword, this identifier may represent a type, and the id\_type field of the Token is set accordingly.

For string literals, a new Token of type STRING is created.

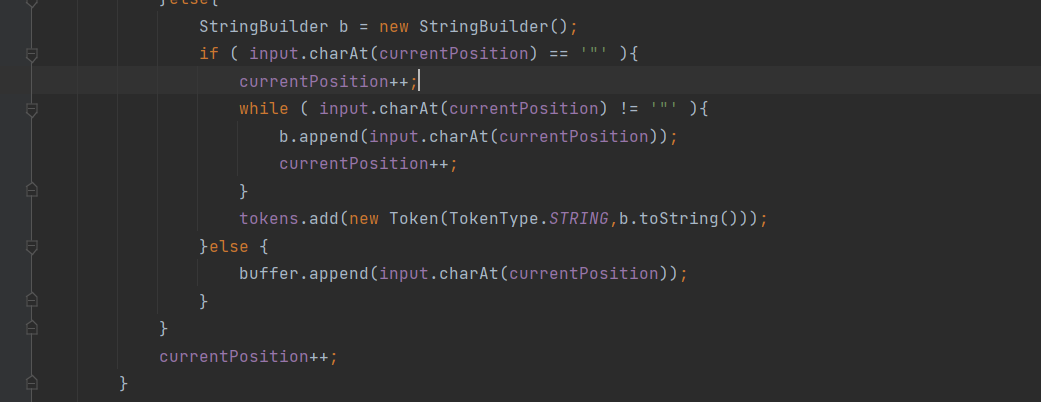
For character literals, a new Token of type CHARACTER is created.

For numeric literals, the method further distinguishes between different number types (decimal, binary, hexadecimal, octal, float) and creates a Token with the corresponding type.

1. Syntax Error Handling: If sbuffer does not match any known pattern, the method prints a syntax error message indicating the unrecognized sequence.

Detailed Explanation of Patterns

1. Identifiers: Matched using the regex [a-zA-Z\_$][a-zA-Z0-9\_$]\*, which corresponds to valid Java identifiers starting with a letter, underscore, or dollar sign, followed by any combination of letters, digits, underscores, or dollar signs.
2. String Literals: Matched using the regex \".\*?\", which captures any sequence of characters enclosed in double quotes by sending a buffer that start with double quotes character and appending upcoming characters until the closed double quotes be captured. Then, send to the to addToken() function to detect it with the above regex as a string literal.



1. Character Literals: Matched using the regex ''(\\.|[^'\\])\*'', which captures any single character enclosed in single quotes.
2. Numeric Literals: Matched using various regex patterns to identify different number formats:

* Decimal: ((\+-)\*\+?|(-\+)\*-?)[1-9][0-9]\*|0
* Binary: ((\+-)\*\+?|(-\+)\*-?)0[bB][01]+
* Hexadecimal: ((\\+-)\*\\+?|(-\\+)\*-?)0[xX][0-9a-fA-F]+")
* Octal: ((\+-)\*\+?|(-\+)\*-?)0[0-7]+
* Float: ((\+-)\*\+?|(-\+)\*-?) [0-9]\*\.?[0-9]+([eE][-+]?[0-9]+)?

1. is\_keyword(String str): This method checks if a given string is a keyword in C.

recognizeOperator(String operator):

The recognizeOperator method is a specialized function within the Lexer class that is responsible for identifying operator symbols and categorizing them into their corresponding TokenType. This method is essential for the correct interpretation of operators during the tokenization process.

The Role of the recognizeOperator Method:

Operators in programming languages are symbols that tell the compiler or interpreter to perform specific mathematical, relational, or logical operations. In the Lexer class, the recognizeOperator method takes a string representing an operator symbol and returns the TokenType that represents the type of operation the symbol corresponds to.

**Process Flow of recognizeOperator:**

* Input Parameter: The method accepts a single parameter, String operator, which is the operator symbol to be recognized.
* Switch Statement: The method uses a switch statement to match the operator parameter against known operator symbols.
* TokenType Assignment: Each case in the switch statement corresponds to a different operator symbol. When a match is found, the method returns the appropriate TokenType that represents that operator.
* Default Case: If the operator does not match any known operator symbols, the method returns TokenType.UnknownOP, indicating an unrecognized operator.Detailed Explanation of Operator Recognition The method recognizes a wide range of operator symbols, including but not limited to:
  + Arithmetic operators: +, -, \*, /, %
  + Increment and decrement operators: ++, --
  + Assignment operators: =, +=, -=, \*=, /=, %=
  + Relational operators: >, <, >=, <=, ==, !=
  + Logical operators: &&, ||, !
  + Bitwise operators: &, |, ^, ~, <<, >>
  + Each operator symbol is associated with a specific TokenType that is used later in the parsing phase to understand the operation to be performed.

#### **Token Class**:

This class represents a token, which is a sequence of characters that have a collective meaning. Each token has a type and a value. The type is determined by the Lexer Class, and the value is the actual text from the input string that the token represents.

#### The Code :

public class Token{

    private TokenType type;

    private String token;

    private String Id\_value;

    private String Id\_type;

    public Token(TokenType type, String token) {

        this.type = type;

        this.token = token;

        this.Id\_value = "";

        this.Id\_type = "";

    }

    public TokenType getType() {

        return type;

    }

    public String getValue() {

        return token;

    }

    public String getId\_value() {

        return Id\_value;

    }

    public void setId\_value(String id\_value) {

        Id\_value = id\_value;

    }

    public String getId\_type() {

        return Id\_type;

    }

    public void setId\_type(String id\_type) {

        Id\_type = id\_type;

    }

    @Override

    public String toString() {

        return "Token{" +

                "type=" + type +

                ", token='" + token + '\'' +

                ", Id\_value='" + Id\_value + '\'' +

                ", Id\_type='" + Id\_type + '\'' +

                '}';

    }

}

#### Code description:

Fields

* type: The type of the token, as defined by the TokenType enum.
* token: The actual text from the input that this token represents.
* Id\_value: The value of the identifier, if this token is an identifier.
* Id\_type: The type of the identifier if this token is an identifier like int, float, in case of functions then return type.

Methods

* Token(TokenType type, String token): This is the constructor for the Token class. It initializes the type, token, Id\_value, and Id\_type.
* getType(), getValue(), getId\_value(), getId\_type(): These are getter methods for the fields of the Token class.
* setId\_value(String id\_value), setId\_type(String id\_type): These are setter methods for the Id\_value and Id\_type fields.
* toString(): This method provides a string representation of the Token object.

### TokenType Class:

This class is an enumeration of the different types of tokens that can be recognized by the Lexer. It includes various categories such as keywords, identifiers, different number types, strings, characters, symbols, and various operators.

#### The code :

package com.example.c\_compiler;

public enum TokenType {

        KEYWORD,

        IDENTIFIER,

        DECIMAL,

        OCTAL,

        BINARY,

        HEX,

        FLOAT,

        STRING,

        Character,

        SYMBOL,

        // Arithmetic Operators

        ADD, SUB, MUL, DIV, MOD, INC, DEC,

        // Assignment Operators

        ASSIGN, ADD\_ASSIGN, SUB\_ASSIGN, MUL\_ASSIGN, DIV\_ASSIGN, MOD\_ASSIGN,

        // Relational Operators

        LT, GT, LE, GE, EQ, NE,

        // Logical Operators

        AND, OR, NOT,

        // Bitwise Operators

        BIT\_AND, BIT\_OR, BIT\_XOR, BIT\_NOT, LEFT\_SHIFT, RIGHT\_SHIFT, UnknownOP

}

#### The code description :

The TokenType enum defines the different types of tokens that can be recognized by the lexer. It includes various categories such as keywords, identifiers, different number types, strings, characters, symbols, and various operators.

## Test Cases :

A screenshot of a computer program

Description automatically generated

A screenshot of a computer program

Description automatically generated

A screen shot of a computer

Description automatically generated

A screen shot of a computer program

Description automatically generatedA screenshot of a computer program

Description automatically generated

A screenshot of a computer program

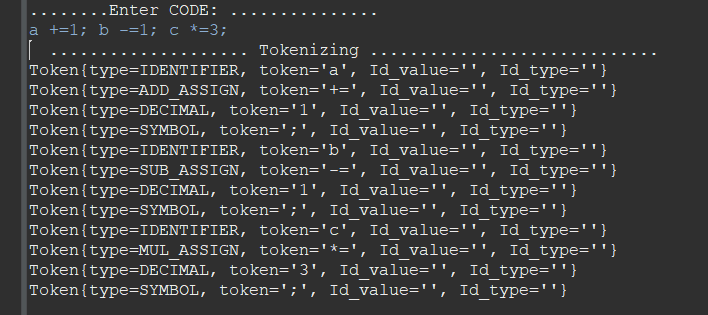
Description automatically generated

A screen shot of a computer program

Description automatically generated

A screenshot of a computer program

Description automatically generated

A screen shot of a computer code

Description automatically generated

A screen shot of a computer program

Description automatically generated

A screen shot of a computer

Description automatically generated

# Symbol table

# SymbolTable Class Documentation

## Overview:

The SymbolTable class is used to manage scopes and symbols in a programming language compiler. It maintains a stack of scopes, with each scope being a map of symbol names to their corresponding `Token` objects.

## Code :

package com.example.c\_compiler;

import java.util.\*;

class SymbolTable {

    private Deque<Map<String, Token>> scopeStack;

    private int current\_scope\_level;

    private List<LinkedList<Map<String, Token>>> allScopes;

    public SymbolTable() {

        allScopes = new ArrayList<>();

        scopeStack = new ArrayDeque<>();

        scopeStack.push(new HashMap<>());

        this.current\_scope\_level = 0;

    }

    public void startScope() {

        scopeStack.push(new HashMap<>());

        current\_scope\_level++;

        // Ensure the allScopes list is large enough

        while (allScopes.size() <= current\_scope\_level) {

            allScopes.add(new LinkedList<>());

        }

    }

    public void endScope() {

        allScopes.get(current\_scope\_level).add(new HashMap<>(scopeStack.peek()));

        scopeStack.pop();

        current\_scope\_level--;

    }

    public boolean addSymbol(String name, Token token) {

        Map<String, Token> currentScope = scopeStack.peek();

        if (currentScope.containsKey(name)) {

            return false;

        } else {

            currentScope.put(name, token);

            return true;

        }

    }

    public void display() {

        // Add the level 0 scope to allScopes

        allScopes.get(0).add(new HashMap<>(scopeStack.peek()));

        System.out.println("---------------------------- Symbol Table ----------------------------");

        System.out.println("-----------------------------------------------------------");

        for (int i = 0; i < allScopes.size(); i++) {

            LinkedList<Map<String, Token>> scopes = allScopes.get(i);

            for (Map<String, Token> scope : scopes) {

                displayScope(i, scope);

            }

        }

    }

    private void displayScope(int scopeIndex, Map<String, Token> scopeSymbols) {

        System.out.println("----------------------- Scope Level " + scopeIndex + " ------------------------");

        for (Map.Entry<String, Token> entry : scopeSymbols.entrySet()) {

            System.out.println("Identifier: " + entry.getKey() + ", Token: " + entry.getValue().getId\_type());

        }

        System.out.println("-----------------------------------------------------------");

    }

}

## Class Variables

- scopeStack: A stack (Deque) that holds the current scope stack. Each scope is a map from symbol names (Strings) to their corresponding `Token` objects.

- current\_scope\_level: An integer that keeps track of the current scope level.

- allScopes: A list of all scopes at each level. Each level is a linked list of maps from symbol names to their corresponding Token objects.

## Methods

* public SymbolTable()**:**
  + This is the constructor of the SymbolTable class. It’s called when a new instance of the class is created. It initializes the scopeStack as an empty stack (Deque) with an empty HashMap pushed onto it. This HashMap represents the global scope (level 0). The current\_scope\_level is set to 0, indicating that we’re currently at the global scope. allScopes is initialized as an empty ArrayList, which will later hold all scopes at each level.
* public voidstartScope()
  + This method is used to start a new scope. It pushes a new, empty HashMap onto the scopeStack, effectively creating a new scope. The current\_scope\_level is incremented by 1, indicating that we’ve moved one level deeper into the scopes. It also ensures that the allScopes list is large enough to hold the new scope level by adding new LinkedLists until its size is greater than or equal to the current\_scope\_level.
* public void endScope()
  + This method is used to end the current scope. It first adds the current scope (the top of the scopeStack) to the allScopes list at the current\_scope\_level. This is done by creating a new HashMap that is a copy of the current scope and adding it to the LinkedList at the current\_scope\_level in allScopes. Then, it pops the current scope off the scopeStack and decrements the current\_scope\_level by 1, indicating that we’ve moved one level up (out of the current scope).
* **public boolean addSymbol(String name, Token token)**
  + This method attempts to add a new symbol to the current scope. It first gets the current scope (the top of the scopeStack). If the symbol (name) already exists in the current scope, it returns false, indicating that the symbol could not be added because it’s already defined in the current scope. Otherwise, it adds the symbol to the current scope and returns true, indicating that the symbol was successfully added.
* public void display()
  + This method is used to print out the symbol table. It first adds the level 0 scope (the global scope) to allScopes. Then, it iterates over all scopes in allScopes. For each scope level, it gets the LinkedList of scopes at that level and iterates over each scope in the LinkedList. It calls displayScope() on each scope to print out the scope.
* private void displayScope(int scopeIndex, Map<String, Token> scopeSymbols)
  + This is a helper method used by display() to print out a single scope. It first prints out the scope level (scopeIndex). Then, it iterates over each symbol in the scope (scopeSymbols). For each symbol, it prints out the symbol name (identifier) and the token associated with the symbol.