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 Punctuation](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 compiler](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 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 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.
* addToken(StringBuilder buffer): This is a helper method that adds a recognized token to the tokens list.
* is\_keyword(String str): This method checks if a given string is a keyword in C.
* recognizeOperator(String operator): This method maps operator symbols to their corresponding token types.

#### **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.

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

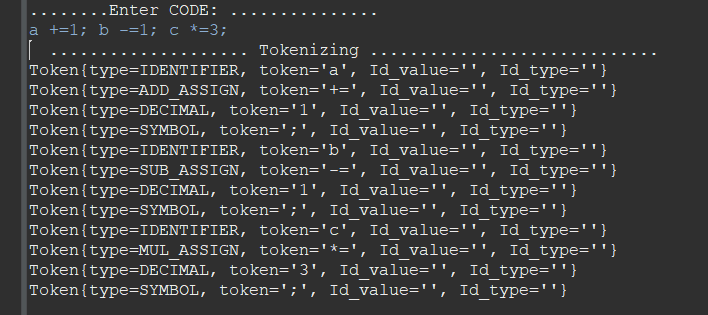
Description automatically generated

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