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**



contents

[Lexer Analyzer 2](#_Toc166318074)

[Overview about Lexer: 2](#_Toc166318075)

[Lexer Responsibility: 3](#_Toc166318076)

[Input and Output 3](#_Toc166318077)

[Regular Languages: 3](#_Toc166318078)

[Tokens 3](#_Toc166318079)

[Tokens : 3](#_Toc166318080)

[Tokens overview: 3](#_Toc166318081)

[Tokens types: 3](#_Toc166318082)

[Lexer Implementation : 4](#_Toc166318083)

[The design of the lexer: 4](#_Toc166318084)

[TokenType Class: 14](#_Toc166318085)

[Symbol table 16](#_Toc166318086)

[Overview: 16](#_Toc166318087)

[Code: 16](#_Toc166318088)

[Class Variables 17](#_Toc166318089)

[Methods 17](#_Toc166318090)

[Test Cases : 19](#_Toc166318091)

[Parser : 29](#_Toc166318092)

[introduction 29](#_Toc166318093)

[Definition 29](#_Toc166318094)

[Responsibilities 29](#_Toc166318095)

[Working Principle 29](#_Toc166318096)

[Implementation of parser class: 30](#_Toc166318097)

[Code description : 61](#_Toc166318098)

# 

# 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:

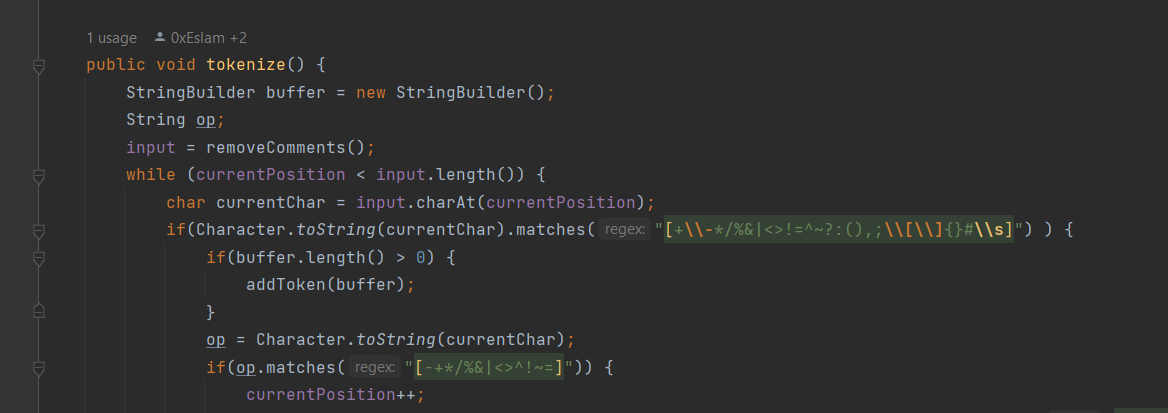
Fields:

* 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.

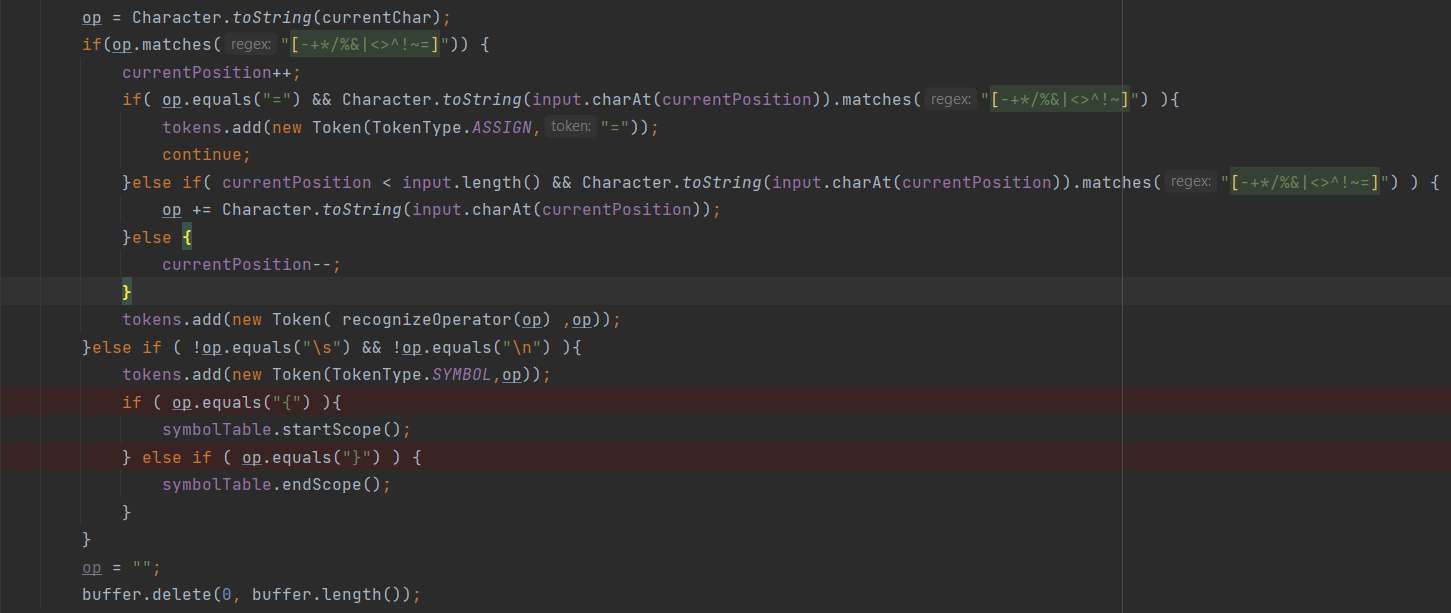
Methods:

* 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\.[0-9]+)|([1-9][0-9]\.?[0-9])|0)([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.

# Symbol table

## Overview:

The Symbol Table 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.

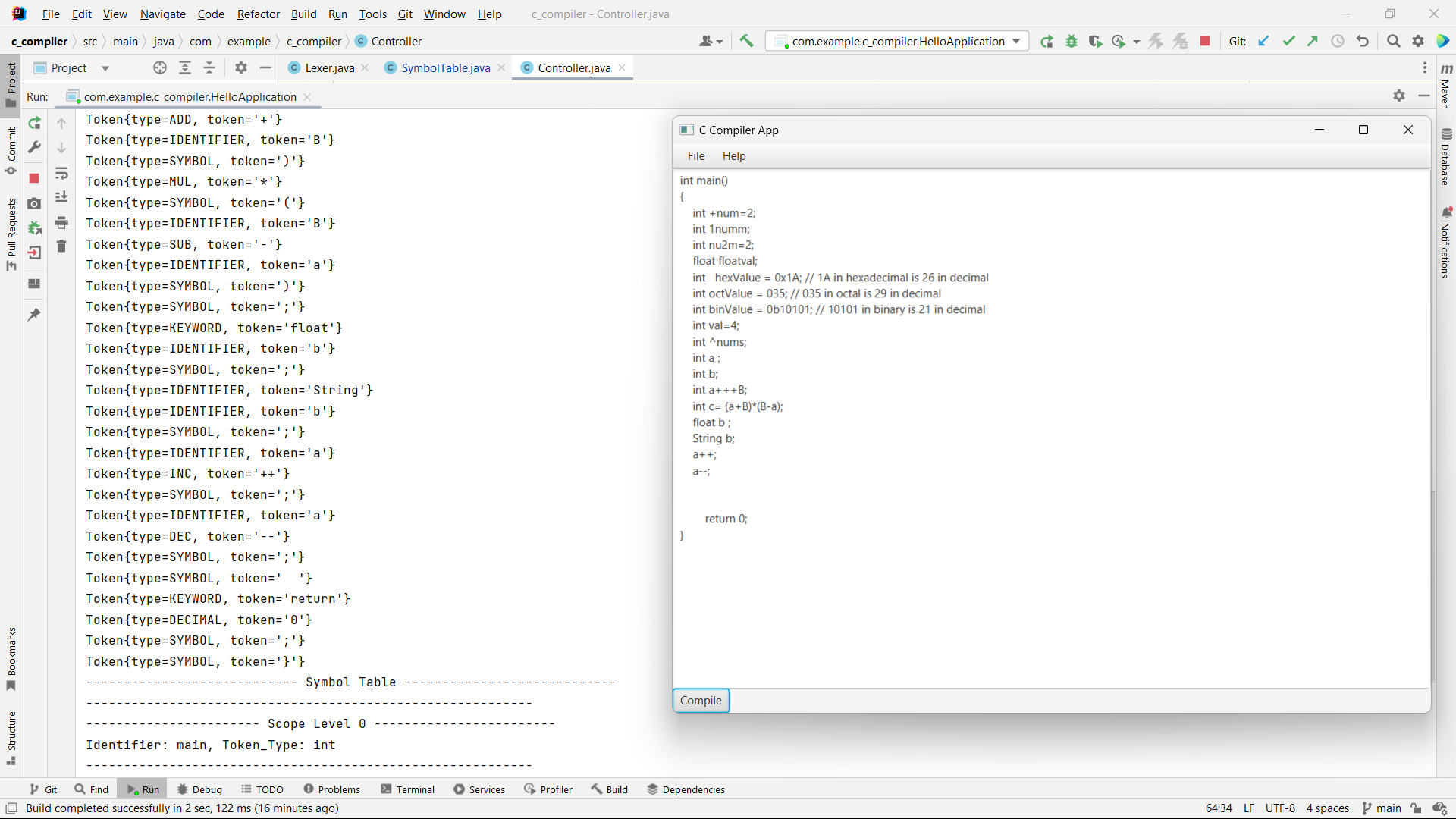
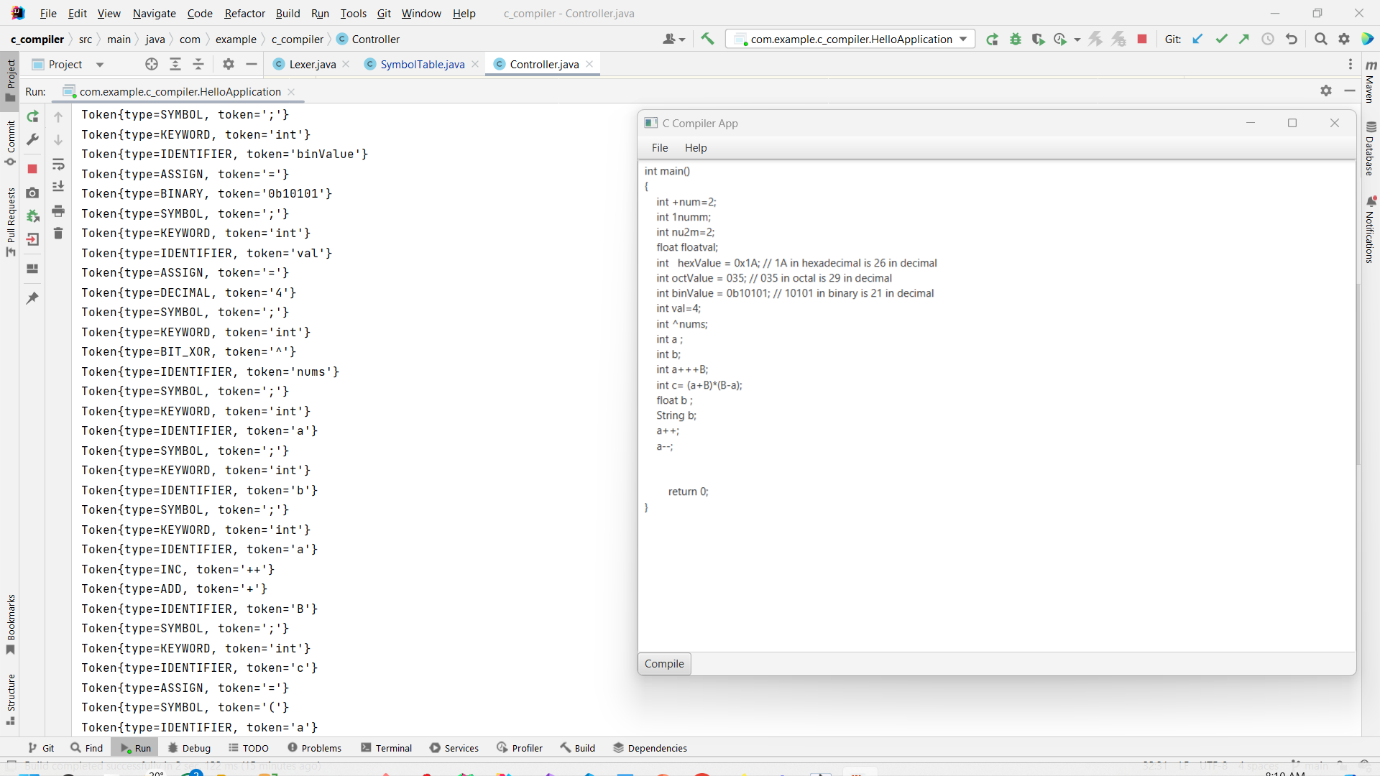
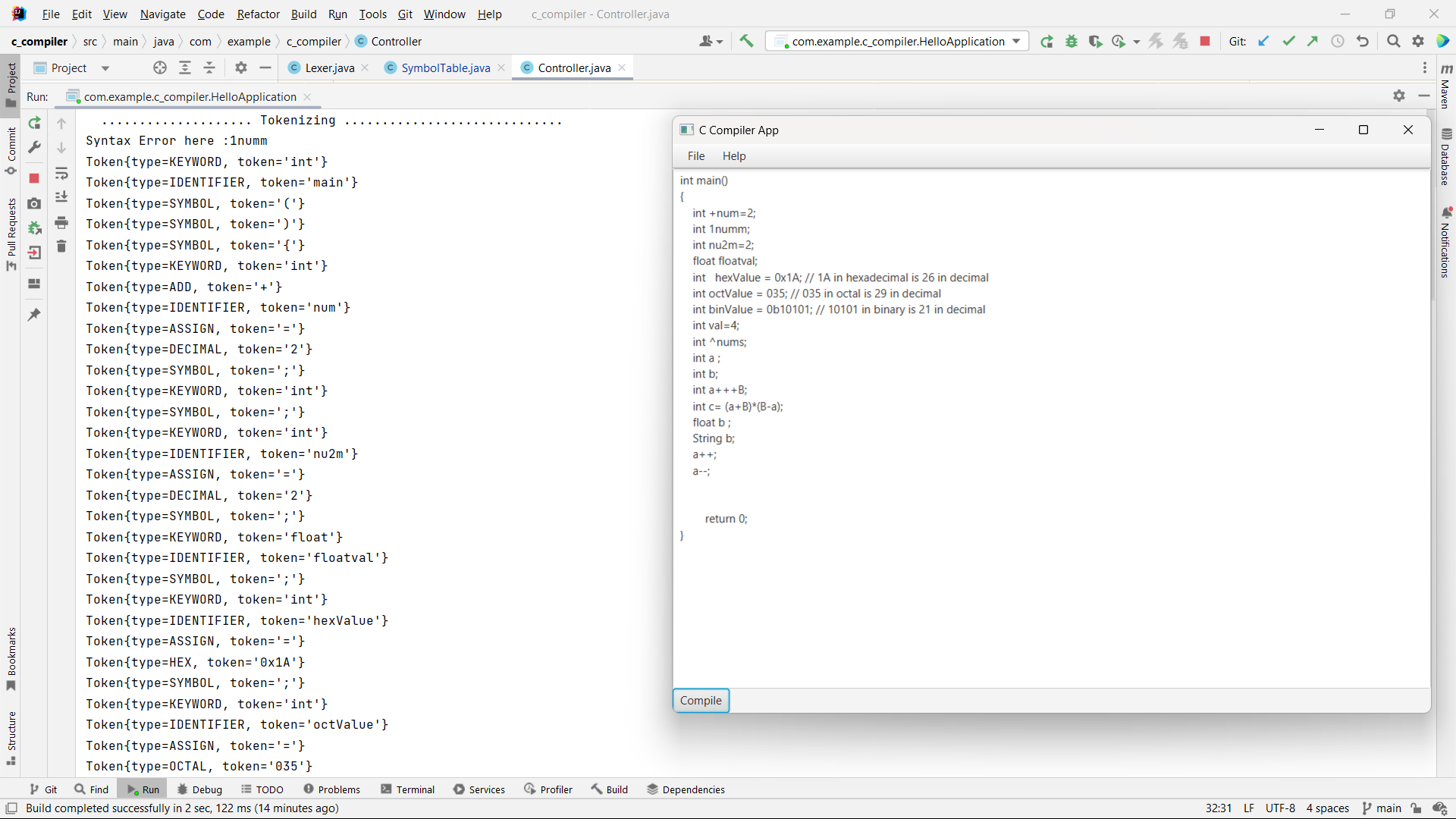
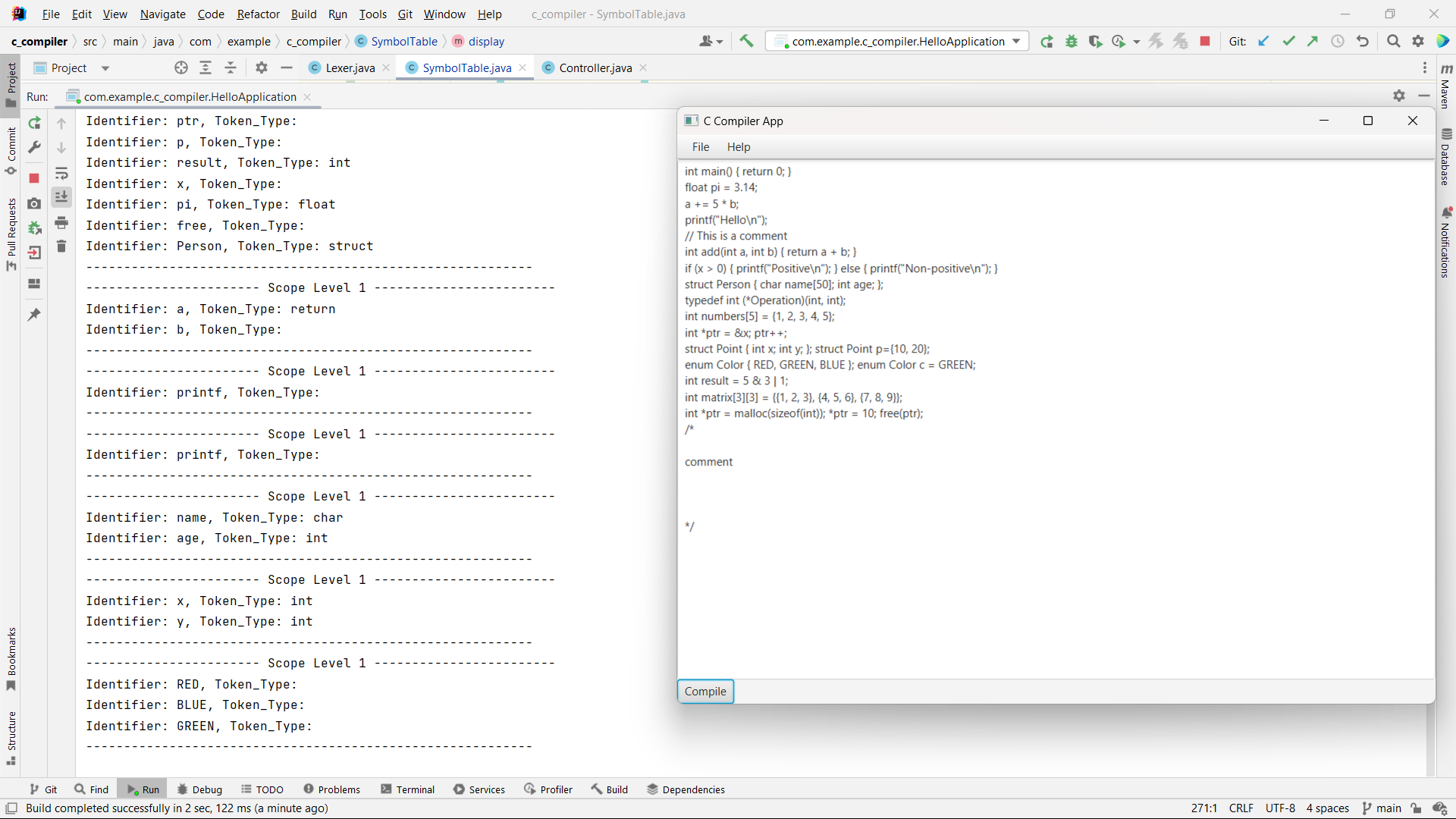
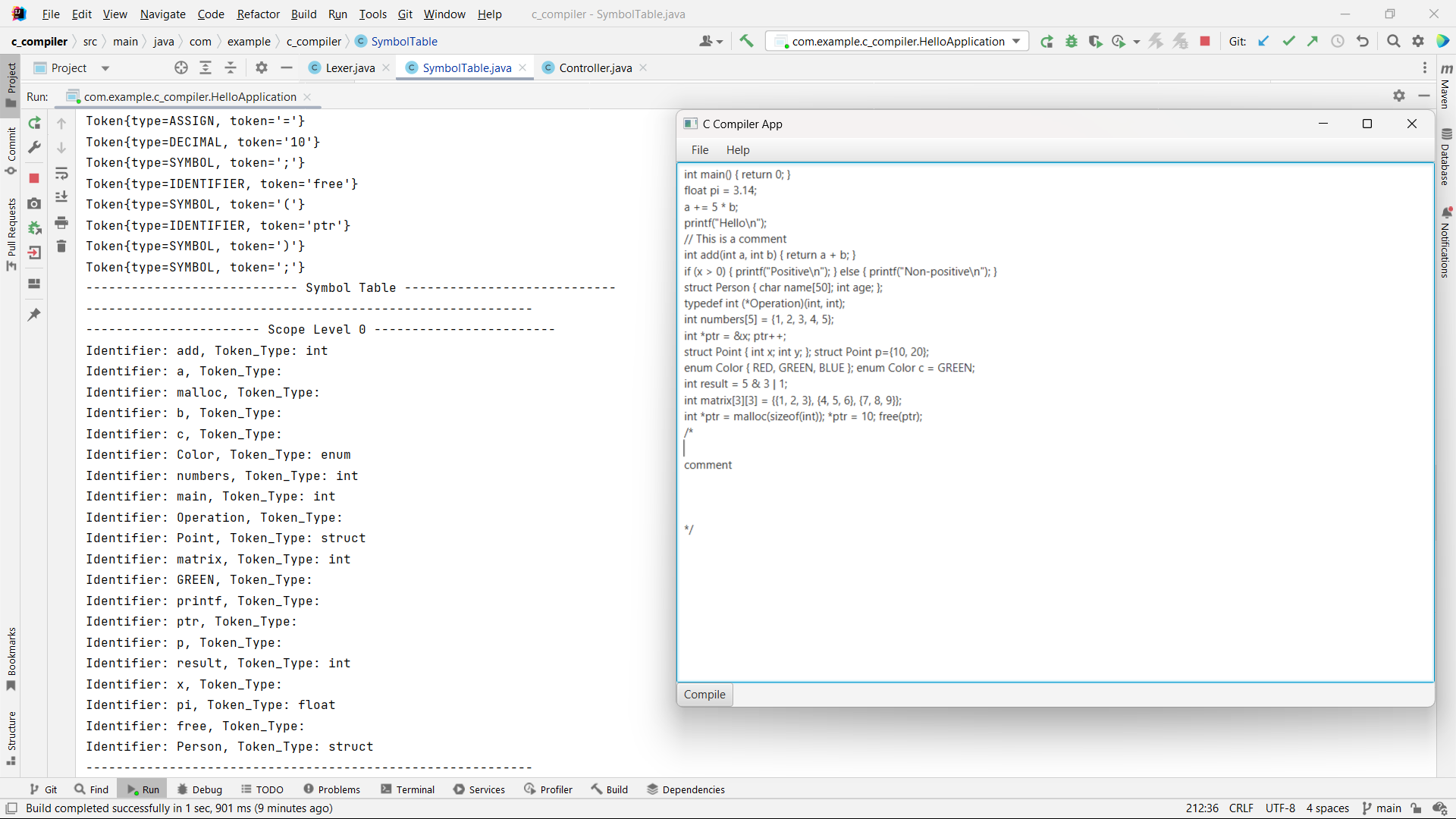
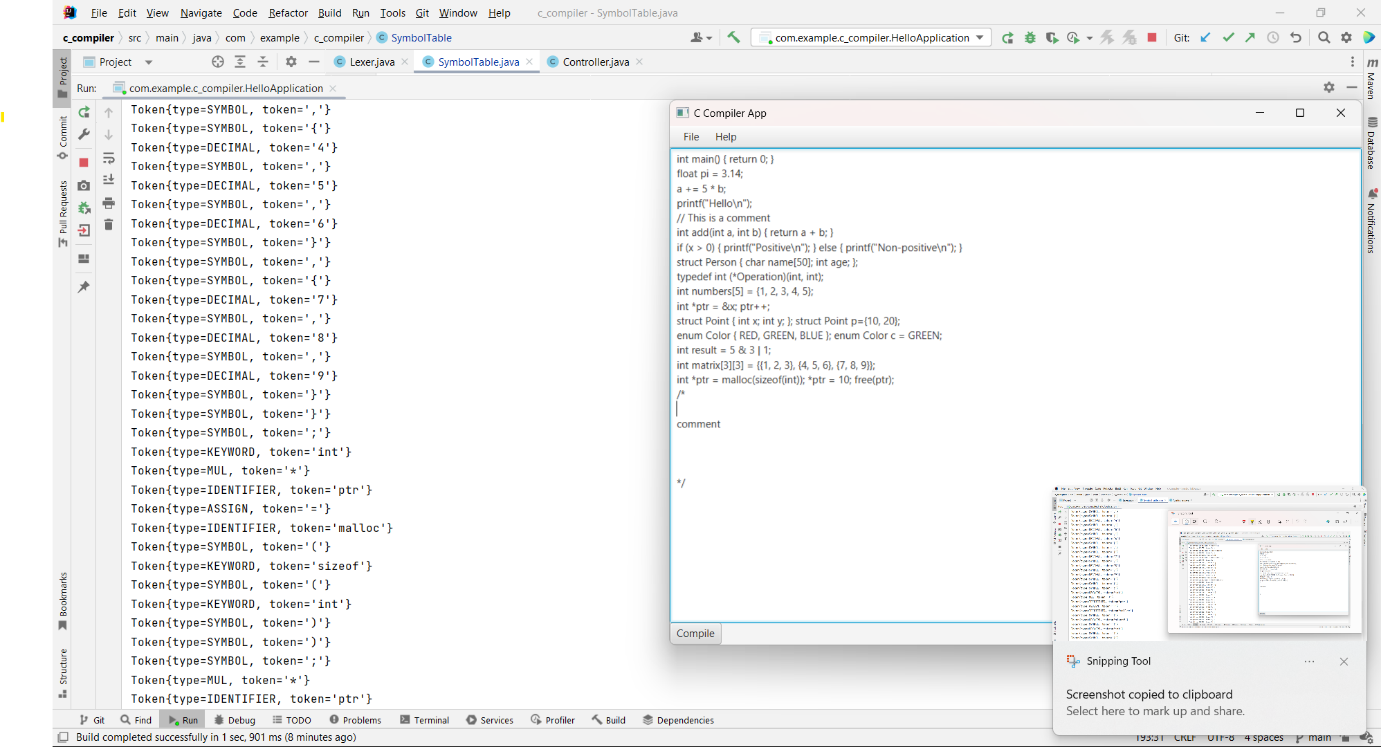
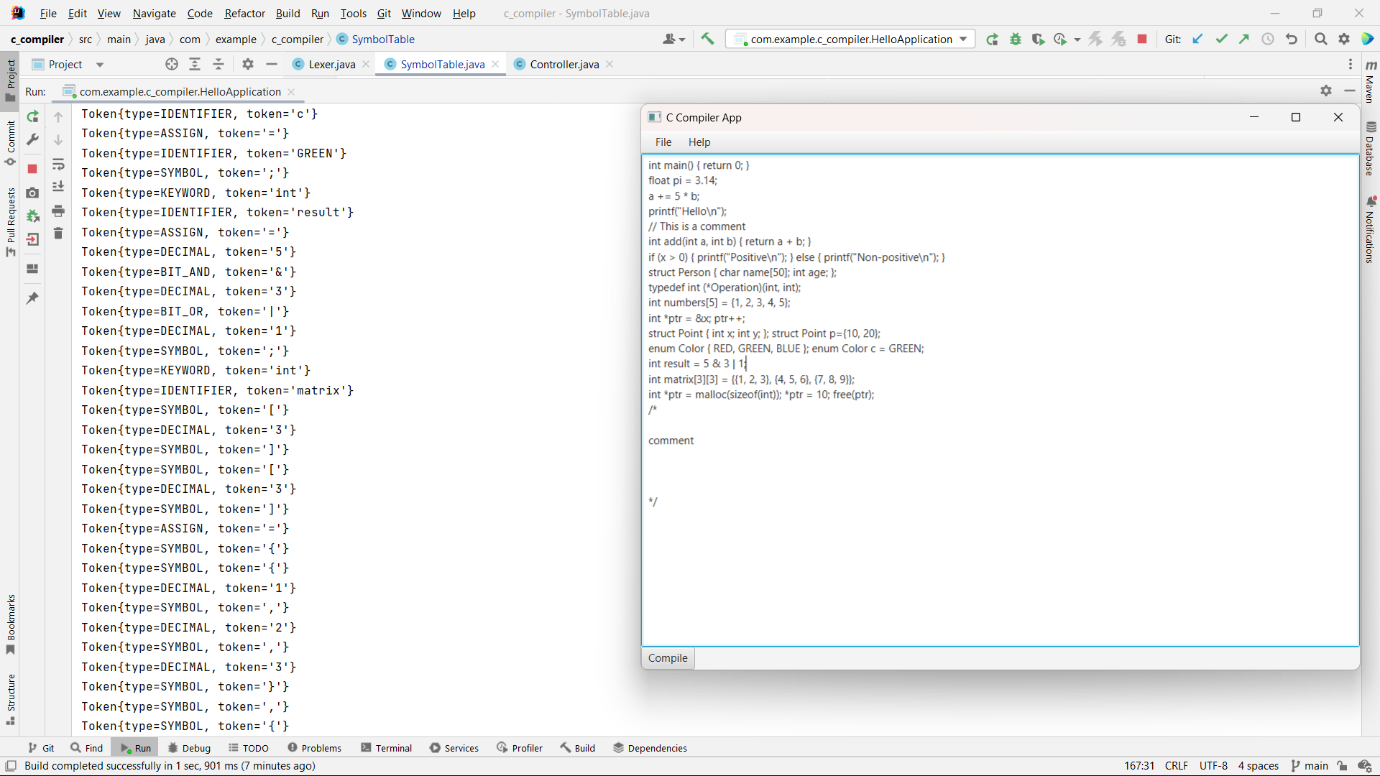
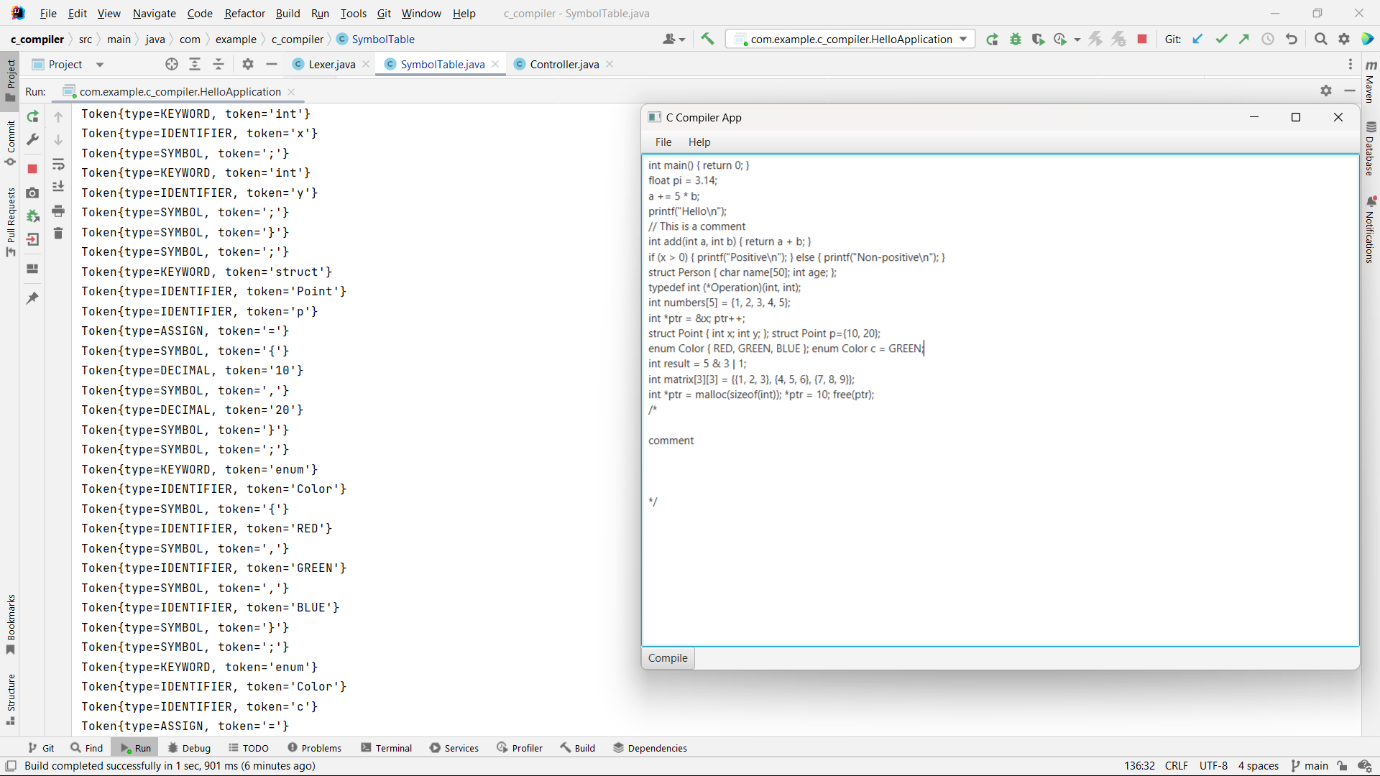
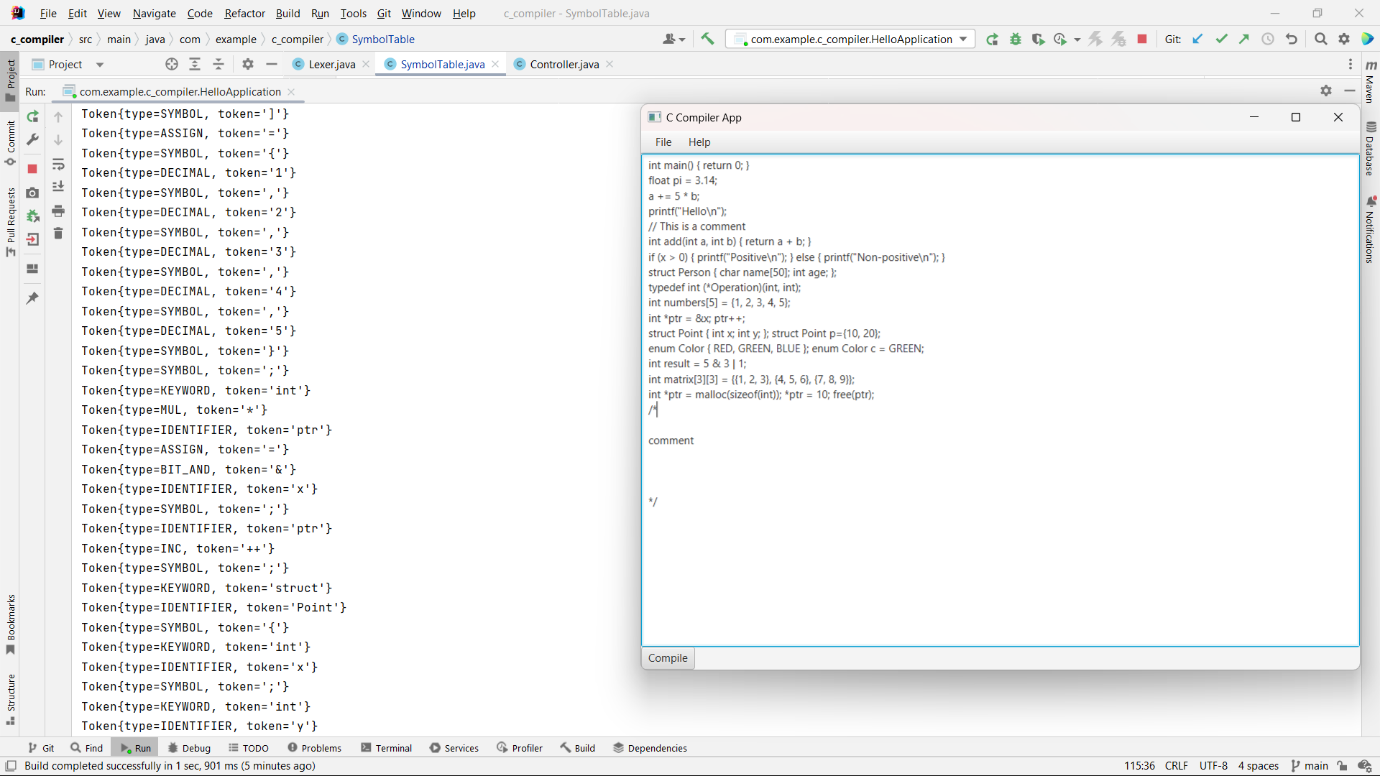
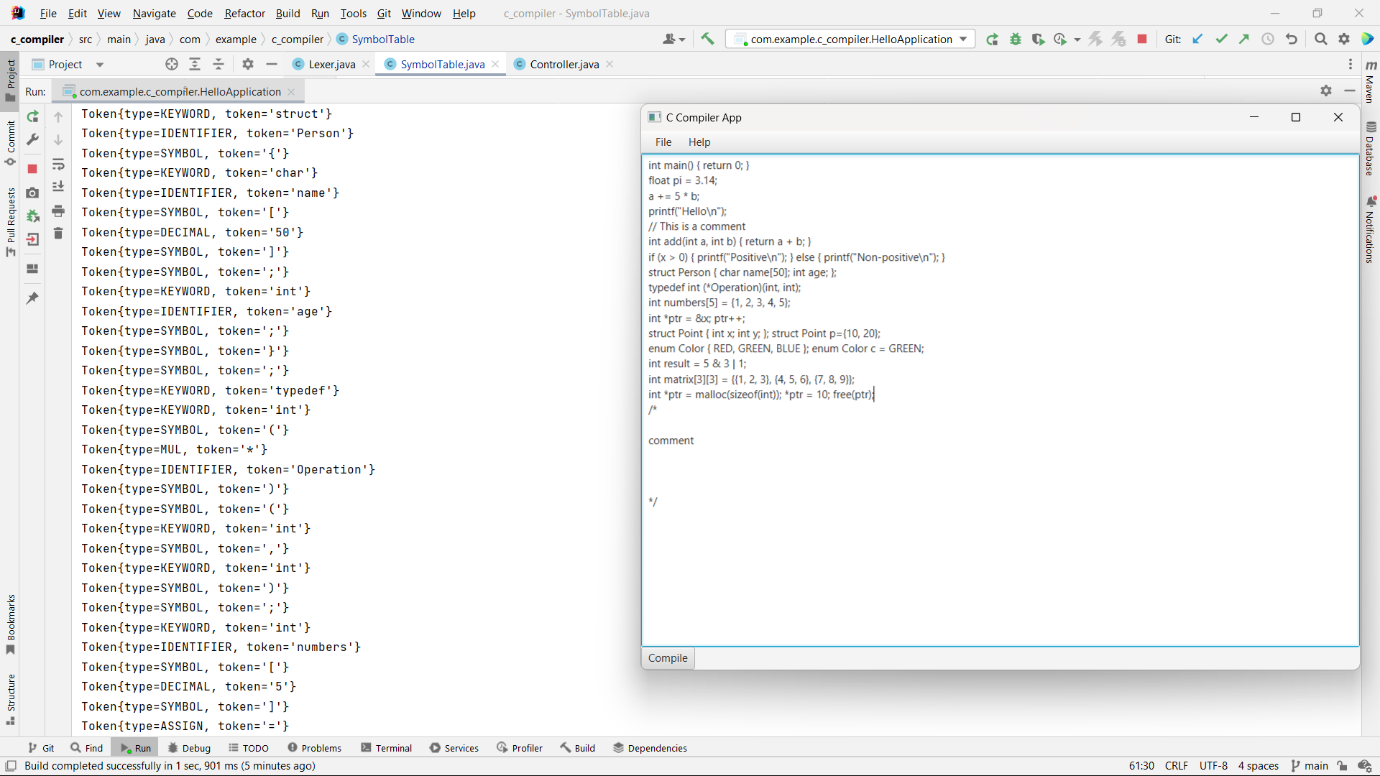
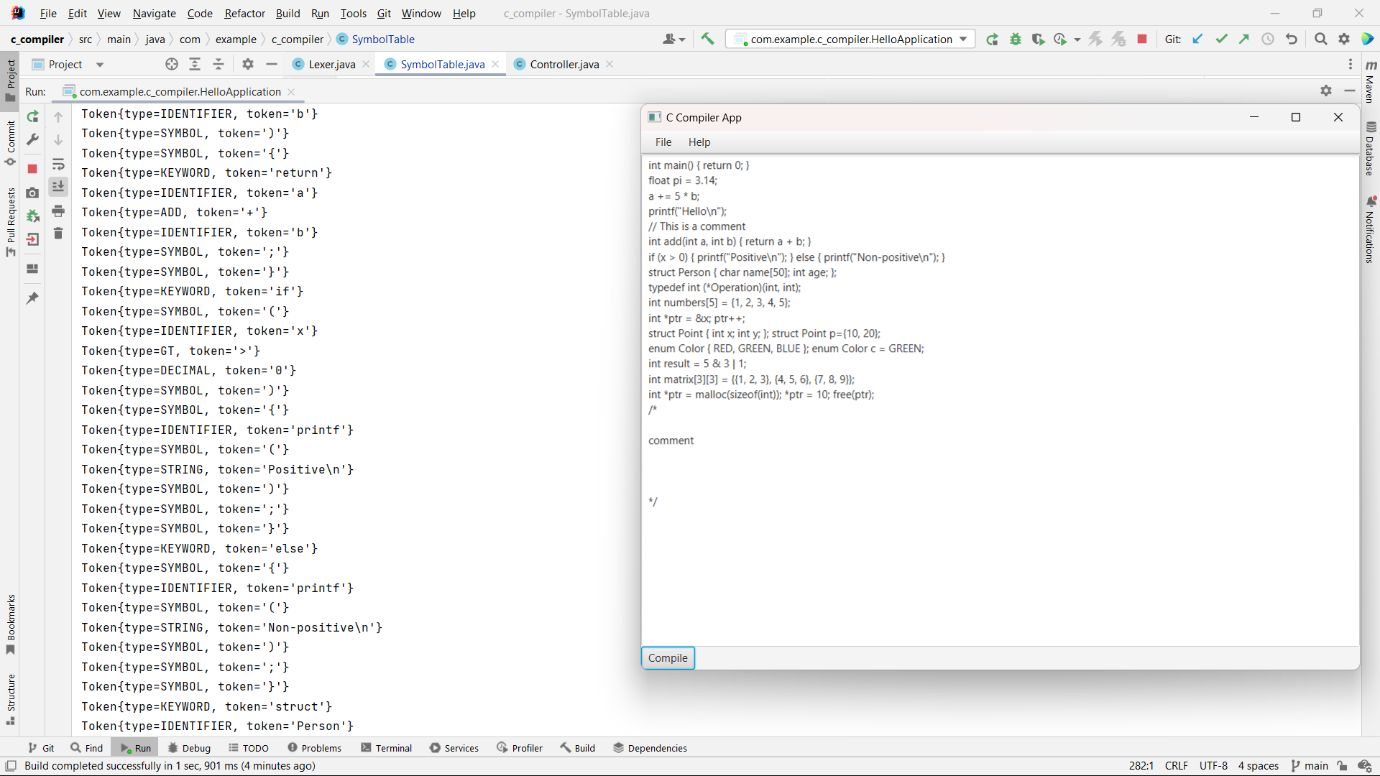
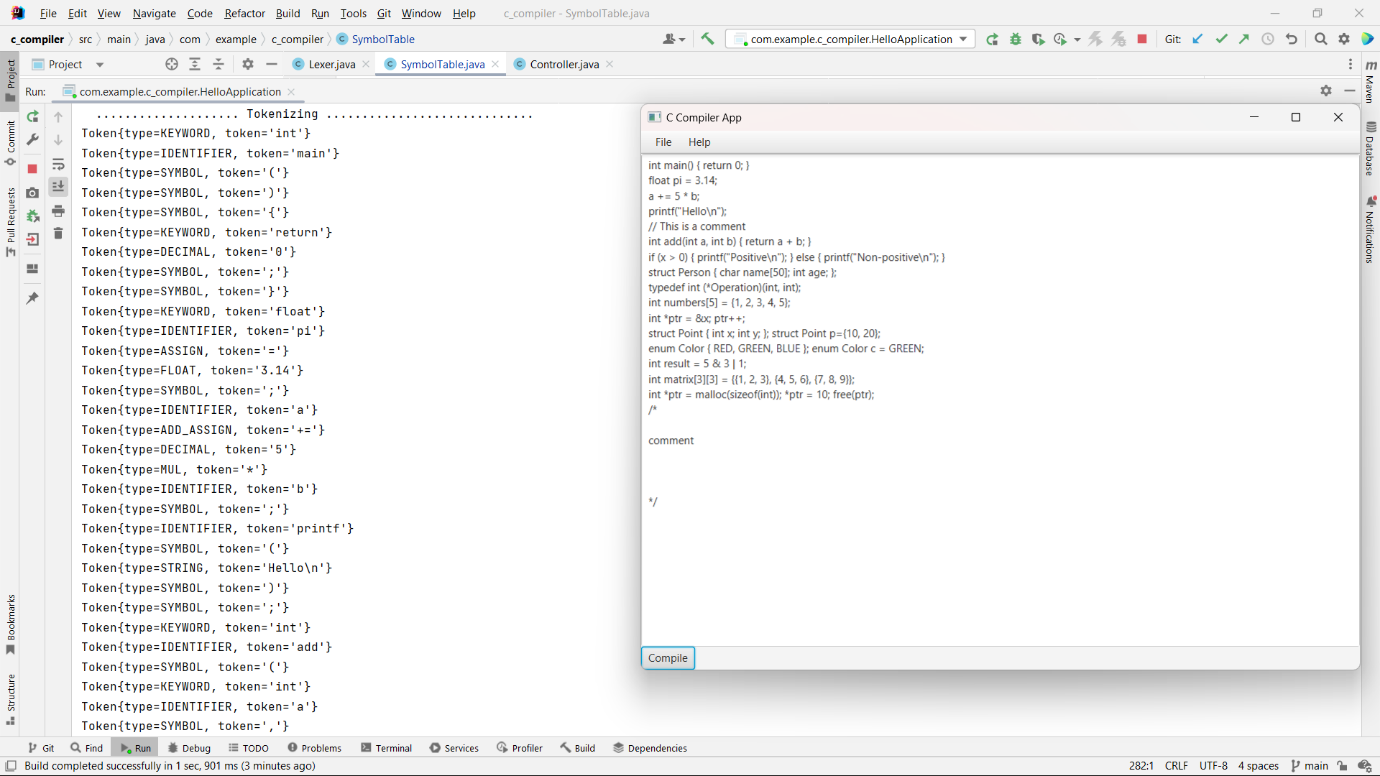
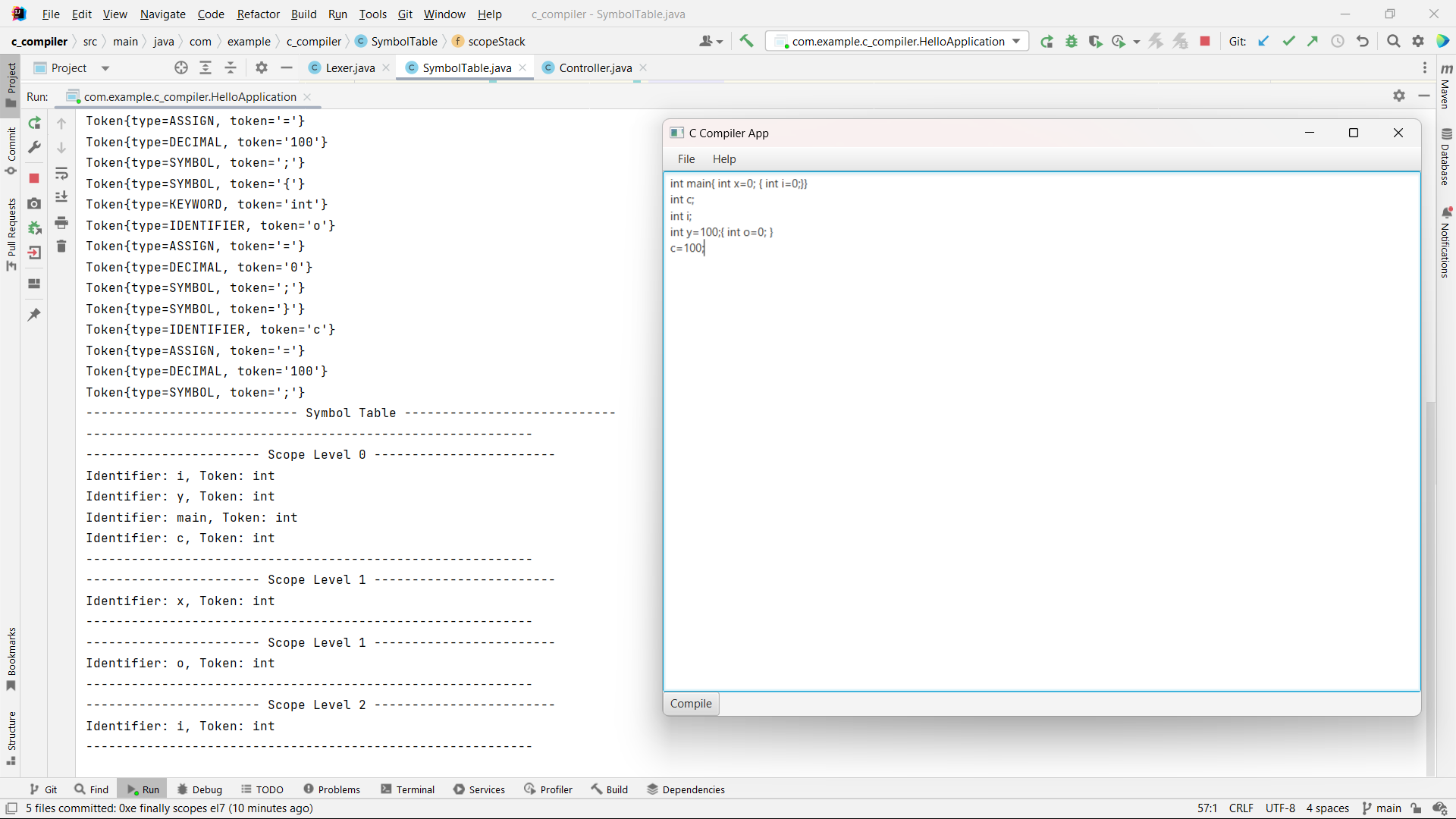
- allScopes: A list of all scopes at each level. Each level is a linked list of maps from symbol names to their corresponding Token objects.

## Methods

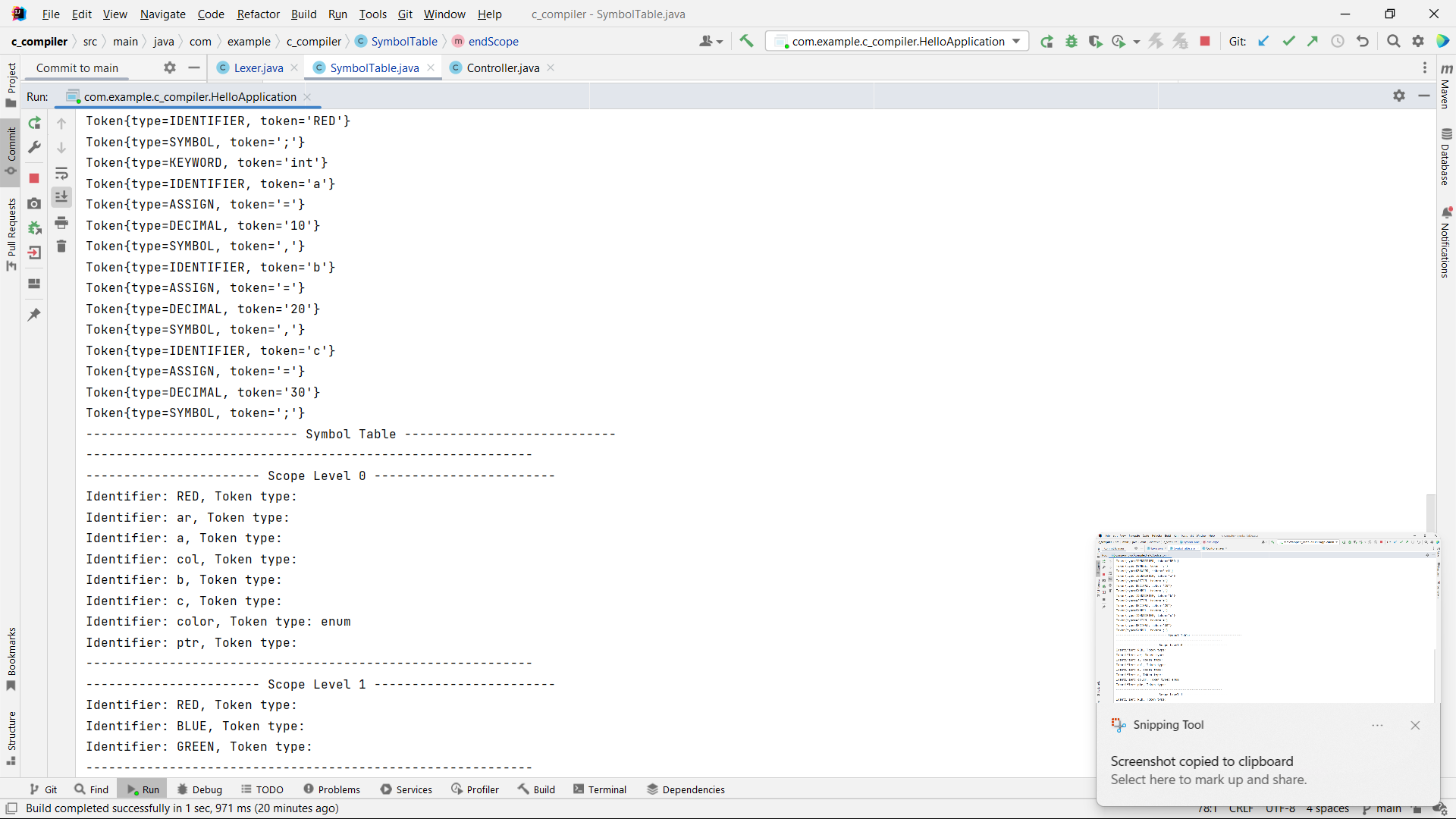
* public SymbolTable()**:**
  + This is the constructor of the SymbolTable class. It’s called when a new instance of the class is created. It initializes the scopeStack as an empty stack (Deque) with an empty HashMap pushed onto it. This HashMap represents the global scope (level 0). The current\_scope\_level 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.

## Test Cases :

A screenshot of a computer

Description automatically generated





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

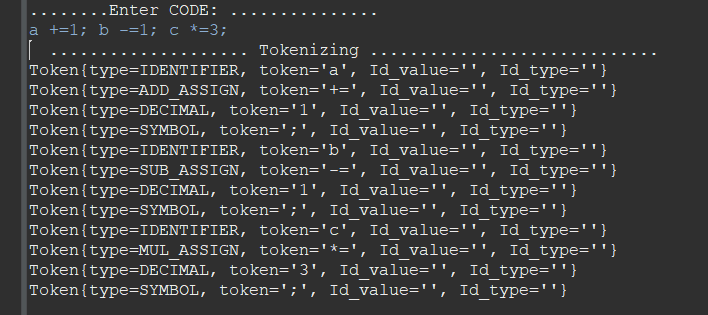
Description automatically generated

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

# Parser :

## introduction

In computer science, particularly in the realm of programming languages, a parser is a crucial component of a compiler or interpreter. Its primary role is to analyze the syntax of a given sequence of tokens (produced by the lexer) to determine its grammatical structure according to a formal grammar specification. This document outlines the definition, responsibilities, and workings of a parser, focusing on its responsibilities following the lexer.

## Definition

A parser, also known as a syntactic analyzer, is a software component responsible for analyzing the structure of a sequence of tokens (typically produced by the lexer) according to a specified grammar. It checks whether the sequence of tokens conforms to the rules defined by the grammar, and if so, it constructs a parse tree or abstract syntax tree (AST) representing the syntactic structure of the input.

## Responsibilities

1. Syntax Analysis

The primary responsibility of the parser is to perform syntax analysis on the stream of tokens produced by the lexer. It ensures that the tokens conform to the rules defined by the grammar of the programming language. This involves determining the hierarchical relationship between tokens and identifying the syntactic constructs such as expressions, statements, declarations, etc.

2. Constructing Parse Trees or ASTs

Upon successful analysis, the parser constructs a parse tree or an abstract syntax tree (AST). A parse tree represents the syntactic structure of the input according to the production rules of the grammar. An AST, on the other hand, abstracts away certain details of the parse tree, focusing more on the essential structure of the program. ASTs are commonly used for further analysis and transformations.

3. Error Handling

Another critical responsibility of the parser is error handling. It detects syntax errors in the input and provides meaningful error messages to aid developers in debugging their code. Error recovery mechanisms may also be employed to continue parsing the input after encountering errors, facilitating the identification of multiple syntax issues in a single compilation pass.

## Working Principle

The parser typically employs parsing techniques such as top-down parsing or bottom-up parsing to analyze the input tokens and construct the parse tree or AST. These techniques involve recursively applying production rules defined by the grammar to reduce the input tokens to higher-level syntactic constructs.

In the context of the provided grammar specification, the parser would start by analyzing the top-level constructs, such as program, function\_declaration, and declaration. It would then recursively delve into each construct, applying the relevant production rules to identify and construct lower-level syntactic elements.

For example, when parsing a function\_declaration, the parser would first identify the type\_specifier, followed by an identifier, and so forth, until it constructs a complete representation of the function declaration.

## Implementation of parser class:

package com.example.c\_compiler;

import java.util.List;

public class Parser {

    private List<Token> tokens;

    private int currentTokenIndex;

    private Node root;

    private Node currentNode;

    String[] dataTypes = {

            "int", "short", "long", "long long", "unsigned int",

            "float", "double", "long double",

            "char", "signed char", "unsigned char",

            "void","char","struct", "enum", "union", "typedef"

    };

    public Parser(List<Token> tokens) {

        this.tokens = tokens;

        this.currentTokenIndex = 0;

        Token program = new Token(TokenType.PROGRAM,"PROGRAM");

        this.root = new Node(program);  // Root node has no token

        this.currentNode = root;

    }

    public boolean parse() {

        return parseProgram();

    }

    private boolean parseProgram() {

        while (currentTokenIndex < tokens.size()) {

            if (!parseFunctionDeclaration() && !parseDeclaration()) {

                return false;

            }

        }

        return true;

    }

    private boolean parseAssignmentExpression() {

        Node assignmentExpressionNode = new Node("assignment\_expression");

        if (!parseConditionalExpression()) {

            return false;

        }

        while (true) {

            if (match("=") || match("+=") || match("-=") || match("\*=") || match("/=") || match("%=") || match("<<=") || match(">>=") || match("&=") || match("^=") || match("|=")) {

                Node operatorNode = new Node(tokens.get(currentTokenIndex)); // Create a node for the operator

                assignmentExpressionNode.addChild(operatorNode); // Add operator node to the assignment expression node

                if (!parseAssignmentExpression()) {

                    return false;

                }

            } else {

                break;

            }

        }

        currentNode.addChild(assignmentExpressionNode); // Add assignment expression node to the current node

        return true;

    }

    private boolean parseConditionalExpression() {

        Node conditionalExpressionNode = new Node("conditional\_expression");

        if (!parseLogicalOrExpression()) {

            return false;

        }

        if (match("?")) {

            Node questionNode = new Node(tokens.get(currentTokenIndex)); // Create a node for the '?'

            conditionalExpressionNode.addChild(questionNode); // Add '?' node to the conditional expression node

            if (!parseExpression()) {

                return false;

            }

            if (!match(":")) {

                reportError("Expected ':' in conditional expression");

                return false;

            }

            Node colonNode = new Node(tokens.get(currentTokenIndex)); // Create a node for the ':'

            conditionalExpressionNode.addChild(colonNode); // Add ':' node to the conditional expression node

            if (!parseConditionalExpression()) {

                return false;

            }

        }

        currentNode.addChild(conditionalExpressionNode); // Add conditional expression node to the current node

        return true;

    }

    private boolean parseLogicalOrExpression() {

        Node logicalOrExpressionNode = new Node("logical\_or\_expression");

        currentNode.addChild(logicalOrExpressionNode); // Add logical or expression node to the current node

        if (parseLogicalAndExpression()) {

            while (match("||")) {

                parseLogicalAndExpression(); // Recursive call

            }

        }

        return true;

    }

    private boolean parseLogicalAndExpression() {

        Node logicalAndExpressionNode = new Node("logical\_and\_expression");

        currentNode.addChild(logicalAndExpressionNode); // Add logical and expression node to the current node

        if (parseInclusiveOrExpression()) {

            while (match("&&")) {

                parseInclusiveOrExpression(); // Recursive call

            }

        }

        return true;

    }

    private boolean parseInclusiveOrExpression() {

        Node inclusiveOrExpressionNode = new Node("inclusive\_or\_expression");

        currentNode.addChild(inclusiveOrExpressionNode); // Add inclusive or expression node to the current node

        if (parseExclusiveOrExpression()) {

            while (match("|")) {

                parseExclusiveOrExpression(); // Recursive call

            }

        }

        return true;

    }

    private boolean parseExclusiveOrExpression() {

        Node exclusiveOrExpressionNode = new Node("exclusive\_or\_expression");

        currentNode.addChild(exclusiveOrExpressionNode); // Add exclusive or expression node to the current node

        if (parseAndExpression()) {

            while (match("^")) {

                parseAndExpression(); // Recursive call

            }

        }

        return true;

    }

    private boolean parseAndExpression() {

        Node andExpressionNode = new Node("and\_expression");

        currentNode.addChild(andExpressionNode); // Add and expression node to the current node

        if (parseEqualityExpression()) {

            while (match("&")) {

                parseEqualityExpression(); // Recursive call

            }

        }

        return true;

    }

    private boolean parseEqualityExpression() {

        Node equalityExpressionNode = new Node("equality\_expression");

        currentNode.addChild(equalityExpressionNode); // Add equality expression node to the current node

        if (parseRelationalExpression()) {

            while (match("==") || match("!=")) {

                parseRelationalExpression(); // Recursive call

            }

        }

        return true;

    }

    private boolean parseShiftExpression() {

        Node shiftExpressionNode = new Node("shift\_expression");

        currentNode.addChild(shiftExpressionNode); // Add shift expression node to the current node

        if (parseAdditiveExpression()) {

            while (match(TokenType.LEFT\_SHIFT) || match(TokenType.RIGHT\_SHIFT)) {

                Node operatorNode = new Node(tokens.get(currentTokenIndex)); // Create a node for the operator

                shiftExpressionNode.addChild(operatorNode); // Add operator node to the shift expression node

                if (!parseAdditiveExpression()) {

                    reportError("Invalid additive expression");

                    return false;

                }

            }

        }

        return true;

    }

    private boolean parseAdditiveExpression() {

        Node additiveExpressionNode = new Node("additive\_expression");

        currentNode.addChild(additiveExpressionNode); // Add additive expression node to the current node

        if (parseMultiplicativeExpression()) {

            while (match(TokenType.ADD) || match(TokenType.SUB)) {

                Node operatorNode = new Node(tokens.get(currentTokenIndex)); // Create a node for the operator

                additiveExpressionNode.addChild(operatorNode); // Add operator node to the additive expression node

                if (!parseMultiplicativeExpression()) {

                    reportError("Invalid multiplicative expression");

                    return false;

                }

            }

        }

        return true;

    }

    private boolean parseMultiplicativeExpression() {

        Node multiplicativeExpressionNode = new Node("multiplicative\_expression");

        currentNode.addChild(multiplicativeExpressionNode); // Add multiplicative expression node to the current node

        if (parseCastExpression()) {

            while (match(TokenType.MUL) || match(TokenType.DIV) || match(TokenType.MOD)) {

                Node operatorNode = new Node(tokens.get(currentTokenIndex)); // Create a node for the operator

                multiplicativeExpressionNode.addChild(operatorNode); // Add operator node to the multiplicative expression node

                if (!parseCastExpression()) {

                    reportError("Invalid cast expression");

                    return false;

                }

            }

        }

        return true;

    }

    private boolean parseCastExpression() {

        Node castExpressionNode = new Node("cast\_expression");

        currentNode.addChild(castExpressionNode); // Add cast expression node to the current node

        if (match("(")) {

            // Check if it's a type name or a unary expression

            if ( is\_a\_type(tokens.get(currentTokenIndex).getValue()) ) {

                // Parse type name

                Node typeNameNode = new Node(tokens.get(currentTokenIndex)); // Create a node for the type name

                castExpressionNode.addChild(typeNameNode); // Add type name node to the cast expression node

                if (!parseTypeName()) {

                    reportError("Invalid type name in cast expression");

                    return false;

                }

                if (!match(")")) {

                    reportError("Expected ')' after type name in cast expression");

                    return false;

                }

                // Parse cast expression

                if (!parseCastExpression()) {

                    reportError("Invalid cast expression");

                    return false;

                }

            } else {

                // Parse unary expression

                if (!parseUnaryExpression()) {

                    reportError("Invalid unary expression in cast expression");

                    return false;

                }

            }

        } else {

            // Parse unary expression

            if (!parseUnaryExpression()) {

                reportError("Invalid unary expression in cast expression");

                return false;

            }

        }

        return true;

    }

    private boolean parseTypeName() {

        Node typeNameNode = new Node("type\_name");

        currentNode.addChild(typeNameNode); // Add type name node to the current node

        // Parse type specifier

        if (!parseTypeSpecifier()) {

            reportError("Invalid type specifier in type name");

            return false;

        }

        // Check for pointer

        while (match("\*")) {

            Node pointerNode = new Node(tokens.get(currentTokenIndex)); // Create a node for the pointer

            typeNameNode.addChild(pointerNode); // Add pointer node to the type name node

            currentTokenIndex++;

        }

        // Parse type qualifier if present

        if (isTypeQualifier()) {

            if (!parseTypeQualifier()) {

                reportError("Invalid type qualifier in type name");

                return false;

            }

        }

        return true;

    }

    private boolean isTypeQualifier() {

        // Check if the current token is a type qualifier

        String tokenValue = tokens.get(currentTokenIndex).getValue();

        return tokenValue.equals("const") || tokenValue.equals("volatile");

    }

    private boolean parseUnaryExpression() {

        Node unaryExpressionNode = new Node("unary\_expression");

        currentNode.addChild(unaryExpressionNode); // Add unary expression node to the current node

        if (parsePostfixExpression()) {

            // Check if there's a unary operator

            if (match(TokenType.ADD) || match(TokenType.SUB) || match(TokenType.NOT) || match(TokenType.BIT\_NOT)) {

                Node operatorNode = new Node(tokens.get(currentTokenIndex)); // Create a node for the operator

                unaryExpressionNode.addChild(operatorNode); // Add operator node to the unary expression node

                if (!parseCastExpression()) {

                    reportError("Invalid cast expression after unary operator");

                    return false;

                }

            }

        } else if (match("sizeof")) {

            Node sizeofNode = new Node(tokens.get(currentTokenIndex)); // Create a node for sizeof

            unaryExpressionNode.addChild(sizeofNode); // Add sizeof node to the unary expression node

            currentTokenIndex++;

            if (match("(")) {

                if ( is\_a\_type(tokens.get(currentTokenIndex).getValue()) ) {

                    // Parse type name

                    if (!parseTypeName()) {

                        reportError("Invalid type name in sizeof expression");

                        return false;

                    }

                    if (!match(")")) {

                        reportError("Expected ')' after type name in sizeof expression");

                        return false;

                    }

                } else {

                    // Parse unary expression

                    if (!parseUnaryExpression()) {

                        reportError("Invalid unary expression in sizeof expression");

                        return false;

                    }

                    if (!match(")")) {

                        reportError("Expected ')' after unary expression in sizeof expression");

                        return false;

                    }

                }

            } else {

                // Parse unary expression

                if (!parseUnaryExpression()) {

                    reportError("Invalid unary expression in sizeof expression");

                    return false;

                }

            }

        } else {

            // Parse postfix expression

            if (!parsePostfixExpression()) {

                reportError("Invalid postfix expression in unary expression");

                return false;

            }

        }

        return true;

    }

    private boolean parseRelationalExpression() {

        Node relationalExpressionNode = new Node("relational\_expression");

        currentNode.addChild(relationalExpressionNode); // Add relational expression node to the current node

        if (parseShiftExpression()) {

            while (match(TokenType.LT) || match(TokenType.GT) || match(TokenType.LE) || match(TokenType.GE)) {

                Node operatorNode = new Node(tokens.get(currentTokenIndex)); // Create a node for the operator

                relationalExpressionNode.addChild(operatorNode); // Add operator node to the relational expression node

                if (!parseShiftExpression()) {

                    reportError("Invalid shift expression");

                    return false;

                }

            }

        }

        return true;

    }

    private boolean parseLabeledStatement() {

        if (match(TokenType.IDENTIFIER)) {

            // Labeled identifier statement

            Node labeledStatementNode = new Node("labeled\_statement");

            currentNode.addChild(labeledStatementNode); // Add labeled statement node to the current node

            Node identifierNode = new Node(tokens.get(currentTokenIndex - 1)); // Create a node for the identifier

            labeledStatementNode.addChild(identifierNode); // Add identifier node to the labeled statement node

            if (!match(":")) {

                reportError("Expected ':' after identifier in labeled statement");

                return false;

            }

            // Parse the statement following the label

            Node statementFollowingLabelNode = new Node("statement\_following\_label");

            if (!parseStatement()) {

                reportError("Invalid statement following label");

                return false;

            }

            labeledStatementNode.addChild(statementFollowingLabelNode);

            return true;

        } else if (match("case")) {

            // Case statement

            Node caseStatementNode = new Node("case\_statement");

            currentNode.addChild(caseStatementNode); // Add case statement node to the current node

            // Parse constant expression after 'case'

            Node constantExpressionNode = new Node(tokens.get(currentTokenIndex));

            if (!parseExpression()) {

                reportError("Invalid constant expression after 'case'");

                return false;

            }

            caseStatementNode.addChild(constantExpressionNode);

            if (!match(":")) {

                reportError("Expected ':' after constant expression in case statement");

                return false;

            }

            // Parse the statement following the case

            Node statementFollowingCaseNode = new Node("statement\_following\_case");

            if (!parseStatement()) {

                reportError("Invalid statement following case");

                return false;

            }

            caseStatementNode.addChild(statementFollowingCaseNode);

            return true;

        } else if (match("default")) {

            // Default statement

            Node defaultStatementNode = new Node("default\_statement");

            currentNode.addChild(defaultStatementNode); // Add default statement node to the current node

            if (!match(":")) {

                reportError("Expected ':' after 'default' in default statement");

                return false;

            }

            // Parse the statement following the default

            Node statementFollowingDefaultNode = new Node("statement\_following\_default");

            if (!parseStatement()) {

                reportError("Invalid statement following default");

                return false;

            }

            defaultStatementNode.addChild(statementFollowingDefaultNode);

            return true;

        } else {

            reportError("Expected labeled statement (identifier, case, or default)" + currentTokenIndex);

            return false;

        }

    }

    private boolean parseStatement() {

        Node statementNode = new Node("statement");

        currentNode.addChild(statementNode); // Add statement node to the current node

        if (parseLabeledStatement()

                || parseCompoundStatement()

                || parseExpressionStatement()

                || parseSelectionStatement()

                || parseIterationStatement()

                || parseJumpStatement()) {

            return true;

        }

        return false;

    }

    private boolean parseIterationStatement() {

        if (match("while")) {

            // While statement

            Node whileStatementNode = new Node("while\_statement");

            currentNode.addChild(whileStatementNode); // Add while statement node to the current node

            if (!match("(")) {

                reportError("Expected '(' after 'while'");

                return false;

            }

            // Parse expression inside parentheses

            Node expressionNode = new Node("expression");

            if (!parseExpression()) {

                reportError("Invalid expression in while statement");

                return false;

            }

            whileStatementNode.addChild(expressionNode);

            if (!match(")")) {

                reportError("Expected ')' after expression in while statement");

                return false;

            }

            // Parse statement inside while loop

            Node statementInsideWhileNode = new Node("statement\_inside\_while");

            if (!parseStatement()) {

                reportError("Invalid statement inside while loop");

                return false;

            }

            whileStatementNode.addChild(statementInsideWhileNode);

            return true;

        } else if (match("do")) {

            // Do-while statement

            Node doWhileStatementNode = new Node("do\_while\_statement");

            currentNode.addChild(doWhileStatementNode); // Add do-while statement node to the current node

            // Parse statement inside do-while loop

            Node statementInsideDoWhileNode = new Node("statement\_inside\_do\_while");

            if (!parseStatement()) {

                reportError("Invalid statement inside do-while loop");

                return false;

            }

            doWhileStatementNode.addChild(statementInsideDoWhileNode);

            if (!match("while")) {

                reportError("Expected 'while' after statement in do-while loop");

                return false;

            }

            if (!match("(")) {

                reportError("Expected '(' after 'while'");

                return false;

            }

            // Parse expression inside parentheses

            Node expressionNode = new Node("expression");

            if (!parseExpression()) {

                reportError("Invalid expression in do-while statement");

                return false;

            }

            doWhileStatementNode.addChild(expressionNode);

            if (!match(")")) {

                reportError("Expected ')' after expression in do-while statement");

                return false;

            }

            if (!match(";")) {

                reportError("Expected ';' after do-while statement");

                return false;

            }

            return true;

        } else if (match("for")) {

            // For statement

            Node forStatementNode = new Node("for\_statement");

            currentNode.addChild(forStatementNode); // Add for statement node to the current node

            if (!match("(")) {

                reportError("Expected '(' after 'for'");

                return false;

            }

            // Parse the first expression statement

            Node expressionStatement1Node = new Node("expression\_statement\_1");

            if (!parseExpressionStatement()) {

                reportError("Invalid expression statement in for loop");

                return false;

            }

            forStatementNode.addChild(expressionStatement1Node);

            // Parse the second expression statement

            Node expressionStatement2Node = new Node("expression\_statement\_2");

            if (!parseExpressionStatement()) {

                reportError("Invalid expression statement in for loop");

                return false;

            }

            forStatementNode.addChild(expressionStatement2Node);

            // Parse the third expression inside for loop

            Node expressionNode = new Node("expression");

            if (!parseExpression()) {

                reportError("Invalid expression in for loop");

                return false;

            }

            forStatementNode.addChild(expressionNode);

            if (!match(")")) {

                reportError("Expected ')' after expression in for loop");

                return false;

            }

            // Parse statement inside for loop

            Node statementInsideForNode = new Node("statement\_inside\_for");

            if (!parseStatement()) {

                reportError("Invalid statement inside for loop");

                return false;

            }

            forStatementNode.addChild(statementInsideForNode);

            return true;

        } else {

            reportError("Expected iteration statement (while, do, or for)");

            return false;

        }

    }

    private boolean parseJumpStatement() {

        if (match("goto")) {

            // Goto statement

            Node gotoStatementNode = new Node("goto\_statement");

            currentNode.addChild(gotoStatementNode); // Add goto statement node to the current node

            if (!match(TokenType.IDENTIFIER)) {

                reportError("Expected identifier after 'goto'");

                return false;

            }

            // Add identifier node to the goto statement node

            Node identifierNode = new Node(tokens.get(currentTokenIndex - 1));

            gotoStatementNode.addChild(identifierNode);

            if (!match(";")) {

                reportError("Expected ';' after goto statement");

                return false;

            }

            return true;

        } else if (match("continue") || match("break")) {

            // Continue or break statement

            Node jumpStatementNode = new Node(tokens.get(currentTokenIndex - 1).getValue() + "\_statement");

            currentNode.addChild(jumpStatementNode); // Add jump statement node to the current node

            if (!match(";")) {

                reportError("Expected ';' after " + tokens.get(currentTokenIndex - 1).getValue() + " statement");

                return false;

            }

            return true;

        } else if (match("return")) {

            // Return statement

            Node returnStatementNode = new Node("return\_statement");

            currentNode.addChild(returnStatementNode); // Add return statement node to the current node

            // Parse expression if present

            if (peek().getType().equals(";")) {

                Node expressionNode = new Node("expression");

                if (!parseExpression()) {

                    reportError("Invalid expression in return statement");

                    return false;

                }

                returnStatementNode.addChild(expressionNode);

            }

            if (!match(";")) {

                reportError("Expected ';' after return statement");

                return false;

            }

            return true;

        } else {

            reportError("Expected jump statement (goto, continue, break, or return)");

            return false;

        }

    }

    private Token peek() {

        int nextIndex = currentTokenIndex + 1;

        if (nextIndex < tokens.size()) {

            return tokens.get(nextIndex);

        }

        return null; // Return null if there are no more tokens to peek

    }

    private boolean parseSelectionStatement() {

        if (match("if")) {

            Node ifStatementNode = new Node("if\_statement");

            currentNode.addChild(ifStatementNode); // Add if statement node to the current node

            if (!match("(")) {

                reportError("Expected '(' after 'if'");

                return false;

            }

            // Parse condition expression

            Node conditionNode = new Node("condition");

            if (!parseExpression()) {

                reportError("Invalid condition expression in if statement");

                return false;

            }

            ifStatementNode.addChild(conditionNode);

            if (!match(")")) {

                reportError("Expected ')' after condition expression in if statement");

                return false;

            }

            // Parse body of if statement

            Node ifBodyNode = new Node("if\_body");

            if (!parseStatement()) {

                reportError("Invalid body of if statement");

                return false;

            }

            ifStatementNode.addChild(ifBodyNode);

            // Check for optional else if and else statements

            if (match("else")) {

                if (match("if")) {

                    // Parse else if statement recursively

                    return parseSelectionStatement();

                } else {

                    // Parse else statement

                    Node elseStatementNode = new Node("else\_statement");

                    if (!parseStatement()) {

                        reportError("Invalid body of else statement");

                        return false;

                    }

                    ifStatementNode.addChild(elseStatementNode);

                }

            }

            return true;

        } else {

            reportError("Expected 'if' to start selection statement");

            return false;

        }

    }

    private boolean parseExpressionStatement() {

        Node expressionStatementNode = new Node("expression\_statement");

        if (parseExpression()) {

            match(";"); // Consume optional semicolon

        } else {

            reportError("Invalid expression in expression statement");

            return false;

        }

        currentNode.addChild(expressionStatementNode); // Add expression statement node to the current node

        return true;

    }

    private boolean parseExpression() {

        Node expressionNode = new Node("expression");

        if (parseAssignmentExpression()) {

            while (match(",")) {

                if (!parseAssignmentExpression()) {

                    return false;

                }

            }

        } else {

            return false;

        }

        currentNode.addChild(expressionNode); // Add expression node to the current node

        return true;

    }

    private boolean parseDeclaration() {

        Node declarationNode = new Node("declaration");

        if (!parseTypeSpecifier()) {

            reportError("Invalid type specifier in declaration");

            return false;

        }

        if (!match(TokenType.IDENTIFIER, declarationNode) || !parseDeclaratorList() || !match(";", declarationNode)) {

            reportError("Invalid declarator or missing semicolon in declaration");

            return false;

        }

        currentNode.addChild(declarationNode); // Add the declaration node to the current node

        return true;

    }

    private boolean parseDeclaratorList() {

        Node declaratorListNode = new Node("declarator\_list");

        if (!parseDeclarator()) {

            return false;

        }

        currentNode.addChild(declaratorListNode); // Add the declarator list node to the current node

        while (match(",")) {

            if (!parseDeclarator()) {

                return false;

            }

        }

        return true;

    }

    private boolean parseDeclarator() {

        Node declaratorNode = new Node("declarator");

        if (match("\*")) {

            declaratorNode.addChild(new Node("\*")); // Add the pointer node to the declarator node

        }

        if (!match(TokenType.IDENTIFIER, declaratorNode)) {

            reportError("Expected identifier in declarator");

            return false;

        }

        if (match("=")) {

            if (!parseInitializer()) {

                reportError("Invalid initializer in declarator");

                return false;

            }

        }

        currentNode.addChild(declaratorNode); // Add the declarator node to the current node

        return true;

    }

    private boolean parseInitializer() {

        Node initializerNode = new Node("initializer");

        if (!parseAssignmentExpression()) {

            reportError("Invalid initializer");

            return false;

        }

        currentNode.addChild(initializerNode); // Add the initializer node to the current node

        return true;

    }

    private boolean parseCompoundStatement() {

        Node compoundStatementNode = new Node("compound\_statement");

        if (!match("{")) {

            reportError("Expected '{' to start compound statement");

            return false;

        }

        Node previousNode = currentNode; // Store the previous current node

        currentNode = compoundStatementNode; // Update the current node

        while (!match("}")) {

            if (!parseDeclaration() && !parseStatement()) {

                reportError("Invalid declaration or statement inside compound statement");

                return false;

            }

        }

        currentNode = previousNode; // Restore the parent node

        currentNode.addChild(compoundStatementNode); // Add the compound statement node to the parent node

        return true;

    }

    private boolean parsePointer() {

        Node pointerNode = new Node("pointer");

        if (match("\*", pointerNode)) {

            while (parseTypeQualifier()) {

                // Parse zero or more type qualifiers

            }

            parsePointer(); // Parse optional pointer

            currentNode.addChild(pointerNode); // Add pointer node to the current node

            return true;

        }

        return false;

    }

    private boolean parseDirectDeclarator() {

        Node directDeclaratorNode = new Node("direct\_declarator");

        if (match(TokenType.IDENTIFIER, directDeclaratorNode)) {

            currentNode.addChild(directDeclaratorNode); // Direct declarator is an identifier

            return true;

        } else if (match("(")) {

            if (!parseDeclarator()) {

                reportError("Invalid declarator inside parentheses");

                return false;

            }

            boolean isParenthesizedDeclarator = match(")", directDeclaratorNode); // Direct declarator is a parenthesized declarator

            if (isParenthesizedDeclarator) {

                currentNode.addChild(directDeclaratorNode); // Add direct declarator node to the current node

            }

            return isParenthesizedDeclarator;

        } else if (parseDirectDeclarator()) {

            if (match("[")) {

                Node constantExpressionNode = new Node("constant\_expression");

                if (!parseConstantExpression()) {

                    reportError("Invalid constant expression inside array declaration");

                    return false;

                }

                directDeclaratorNode.addChild(constantExpressionNode); // Add constant expression node

                boolean isArray = match("]", directDeclaratorNode); // Direct declarator is an array

                if (isArray) {

                    currentNode.addChild(directDeclaratorNode); // Add direct declarator node to the current node

                }

                return isArray;

            } else if (match("(")) {

                if (match(")", directDeclaratorNode)) {

                    currentNode.addChild(directDeclaratorNode); // Direct declarator is a function with no parameters

                    return true;

                }

                Node parameterList = new Node("parameter\_list");

                if (!parseIdentifierList()) {

                    reportError("Invalid parameter list in function declaration");

                    return false;

                }

                directDeclaratorNode.addChild(parameterList); // Add parameter list node

                boolean isFunctionWithParameters = match(")", directDeclaratorNode); // Direct declarator is a function with parameters

                if (isFunctionWithParameters) {

                    currentNode.addChild(directDeclaratorNode); // Add direct declarator node to the current node

                }

                return isFunctionWithParameters;

            }

        }

        reportError("Invalid direct declarator");

        return false;

    }

    private boolean parseConstantExpression() {

        Node constantExpressionNode = new Node("constant\_expression");

        // Check if the current token represents a constant

        TokenType tokenType = tokens.get(currentTokenIndex).getType();

        if (tokenType == TokenType.DECIMAL ||

                tokenType == TokenType.OCTAL ||

                tokenType == TokenType.BINARY ||

                tokenType == TokenType.HEX ||

                tokenType == TokenType.FLOAT ||

                tokenType == TokenType.STRING ||

                tokenType == TokenType.Character) {

            // Create a node for the constant and add it to the constant expression node

            Node constantNode = new Node(tokens.get(currentTokenIndex).getValue());

            constantExpressionNode.addChild(constantNode);

            currentTokenIndex++; // Move to the next token

            currentNode.addChild(constantExpressionNode); // Add constant expression node to the current node

            return true; // Parsing successful

        } else {

            reportError("Invalid constant expression");

            return false; // Parsing failed

        }

    }

    private boolean parseIdentifierList() {

        Node identifierListNode = new Node("identifier\_list");

        if (!match(TokenType.IDENTIFIER, identifierListNode)) {

            reportError("Expected identifier");

            return false;

        }

        while (match(",")) {

            if (!match(TokenType.IDENTIFIER, identifierListNode)) {

                reportError("Expected identifier after comma");

                return false;

            }

        }

        currentNode.addChild(identifierListNode); // Add identifier list node to the current node

        return true;

    }

    private boolean parseTypeSpecifier() {

        Node typeSpecifierNode = new Node("type\_specifier");

        if (match(TokenType.KEYWORD, typeSpecifierNode)) {

            currentNode.addChild(typeSpecifierNode); // Add type specifier node to the current node

            return true;

        } else if (parseStructOrUnionSpecifier() || parseEnumSpecifier() || match(TokenType.IDENTIFIER, typeSpecifierNode)) {

            currentNode.addChild(typeSpecifierNode); // Add type specifier node to the current node

            return true;

        }

        reportError("Invalid type specifier");

        return false;

    }

    private boolean parseStructOrUnionSpecifier() {

        Node structOrUnionSpecifierNode = new Node("struct\_or\_union\_specifier");

        if (!match("struct", structOrUnionSpecifierNode) && !match("union", structOrUnionSpecifierNode)) {

            //reportError("Expected 'struct' or 'union'");

            return false;

        }

        if (match(TokenType.IDENTIFIER, structOrUnionSpecifierNode)) {

            // Add identifier node if present

            structOrUnionSpecifierNode.addChild(new Node(tokens.get(currentTokenIndex - 1).getValue()));

        }

        currentNode.addChild(structOrUnionSpecifierNode); // Add struct or union specifier node to the current node

        if (!match("{", structOrUnionSpecifierNode)) {

            reportError("Expected '{' after struct or union specifier");

            return false;

        }

        Node structDeclarationListNode = new Node("struct\_declaration\_list");

        if (!parseStructDeclarationList()) {

            return false;

        }

        structOrUnionSpecifierNode.addChild(structDeclarationListNode); // Add struct declaration list node

        if (!match("}", structOrUnionSpecifierNode)) {

            reportError("Expected '}' after struct or union declaration list");

            return false;

        }

        return true;

    }

    private boolean parseStructDeclarationList() {

        Node structDeclarationListNode = new Node("struct\_declaration\_list");

        while (true) {

            Node structDeclarationNode = new Node("struct\_declaration");

            if (!parseSpecifierQualifierList()) {

                reportError("Invalid specifier qualifier list in struct declaration");

                return false;

            }

            if (!parseStructDeclaratorList()) {

                reportError("Invalid struct declarator list in struct declaration");

                return false;

            }

            structDeclarationListNode.addChild(structDeclarationNode); // Add struct declaration node

            if (!match(";")) {

                reportError("Expected ';' after struct declaration");

                return false;

            }

            if (match(";")) {

                break; // End of struct declaration list

            }

        }

        currentNode.addChild(structDeclarationListNode); // Add struct declaration list node to the current node

        return true;

    }

    private boolean parseStructDeclaration() {

        Node structDeclarationNode = new Node("struct\_declaration");

        if (!match(";")) {

            reportError("Expected ';' in struct declaration");

            return false;

        }

        currentNode.addChild(structDeclarationNode); // Add struct declaration node to the current node

        return true;

    }

    private boolean parseStructDeclaratorList() {

        Node structDeclaratorListNode = new Node("struct\_declarator\_list");

        if (!parseStructDeclarator()) {

            reportError("Invalid struct declarator in struct declarator list");

            return false;

        }

        currentNode.addChild(structDeclaratorListNode); // Add struct declarator list node to the current node

        while (match(",")) {

            if (!parseStructDeclarator()) {

                reportError("Invalid struct declarator in struct declarator list");

                return false;

            }

        }

        return true;

    }

    private boolean parseSpecifierQualifierList() {

        Node specifierQualifierListNode = new Node("specifier\_qualifier\_list");

        if (parseTypeSpecifier() || parseTypeQualifier()) {

            if (parseSpecifierQualifierList()) {

                currentNode.addChild(specifierQualifierListNode); // Add specifier qualifier list node to the current node

                return true;

            }

            return true; // Type specifier/qualifier without a specifier qualifier list

        }

        reportError("Invalid specifier qualifier list");

        return false;

    }

    private boolean parseStructDeclarator() {

        Node structDeclaratorNode = new Node("struct\_declarator");

        if (parseDeclarator()) {

            if (match(":")) {

                Node constantExpressionNode = new Node("constant\_expression");

                if (!parseConstantExpression()) {

                    reportError("Invalid constant expression in struct declarator");

                    return false;

                }

                structDeclaratorNode.addChild(constantExpressionNode); // Add constant expression node

            }

            currentNode.addChild(structDeclaratorNode); // Add struct declarator node to the current node

            return true;

        }

        reportError("Invalid struct declarator");

        return false;

    }

    private boolean parseTypeQualifier() {

        if (match("const") || match("volatile")) {

            currentNode.addChild(new Node(tokens.get(currentTokenIndex - 1).getValue())); // Add type qualifier node to the current node

            return true;

        }

        reportError("Expected 'const' or 'volatile' as type qualifier");

        return false;

    }

    private boolean parseEnumSpecifier() {

        Node enumSpecifierNode = new Node("enum\_specifier");

        if (!match("enum", enumSpecifierNode)) {

            //reportError("Expected 'enum' in enum specifier");

            return false;

        }

        if (match(TokenType.IDENTIFIER, enumSpecifierNode)) {

            enumSpecifierNode.addChild(new Node(tokens.get(currentTokenIndex - 1).getValue())); // Add identifier node if present

        }

        currentNode.addChild(enumSpecifierNode); // Add enum specifier node to the current node

        if (!match("{")) {

            reportError("Expected '{' after enum specifier");

            return false;

        }

        Node enumeratorListNode = new Node("enumerator\_list");

        if (!parseEnumeratorList()) {

            return false;

        }

        enumSpecifierNode.addChild(enumeratorListNode); // Add enumerator list node

        if (!match("}")) {

            reportError("Expected '}' after enum declaration");

            return false;

        }

        return true;

    }

    private boolean parseEnumeratorList() {

        Node enumeratorListNode = new Node("enumerator\_list");

        Node enumeratorNode = new Node("enumerator");

        if (!parseEnumerator()) {

            return false;

        }

        enumeratorListNode.addChild(enumeratorNode); // Add enumerator node

        while (match(",")) {

            if (!parseEnumerator()) {

                return false;

            }

        }

        currentNode.addChild(enumeratorListNode); // Add enumerator list node to the current node

        return true;

    }

    private boolean parseEnumerator() {

        Node enumeratorNode = new Node("enumerator");

        if (!match(TokenType.IDENTIFIER, enumeratorNode)) {

            reportError("Expected identifier in enumerator declaration");

            return false;

        }

        if (match("=")) {

            Node constantExpressionNode = new Node("constant\_expression");

            if (!parseConstantExpression()) {

                reportError("Expected constant expression after '=' in enumerator declaration");

                return false;

            }

            enumeratorNode.addChild(constantExpressionNode); // Add constant expression node

        }

        currentNode.addChild(enumeratorNode); // Add enumerator node to the current node

        return true;

    }

    private boolean parseInitializerList() {

        Node initializerListNode = new Node("initializer\_list");

        Node initializerNode = new Node("initializer");

        if (!parseInitializer()) {

            reportError("Invalid initializer in initializer list");

            return false;

        }

        initializerListNode.addChild(initializerNode); // Add initializer node

        while (match(",")) {

            if (!parseInitializer()) {

                reportError("Invalid initializer in initializer list");

                return false;

            }

        }

        currentNode.addChild(initializerListNode); // Add initializer list node to the current node

        return true;

    }

    private boolean parseParameterDeclaration() {

        Node parameterNode = new Node("parameter\_declaration");

        if (!parseTypeSpecifier() || !match(TokenType.IDENTIFIER, parameterNode)) {

            reportError("Invalid parameter declaration");

            return false;

        }

        currentNode.addChild(parameterNode); // Add the parameter node to the current node

        return true;

    }

    private boolean parseFunctionDeclaration() {

        Node functionDeclarationNode = new Node("function\_declaration");

        int startIndex = currentTokenIndex; // Record the starting index of the function declaration

        if (!parseTypeSpecifier() || !match(TokenType.IDENTIFIER, functionDeclarationNode) || !match("(", functionDeclarationNode)) {

            reportError("Invalid function declaration");

            currentTokenIndex = startIndex; // Reset the current token index if parsing fails

            return false;

        }

        // Parse parameter list if present

        if (!match(")")) {

            if (!parseParameterList()) {

                reportError("Invalid parameter list in function declaration");

                currentTokenIndex = startIndex; // Reset the current token index if parsing fails

                return false;

            }

            if (!match(")")) {

                reportError("Expected ')' after parameter list in function declaration");

                currentTokenIndex = startIndex; // Reset the current token index if parsing fails

                return false;

            }

        }

        // Parse compound statement

        if (!parseCompoundStatement()) {

            reportError("Invalid compound statement in function declaration");

            currentTokenIndex = startIndex; // Reset the current token index if parsing fails

            return false;

        }

        currentNode.addChild(functionDeclarationNode); // Add the function declaration node to the current node

        return true;

    }

    private boolean parseParameterList() {

        // Create a node for the parameter list

        Node parameterListNode = new Node("parameter\_list");

        // Add the parameter list node to the current node

        currentNode.addChild(parameterListNode);

        // Parse the first parameter declaration, if present

        if (!parseParameterDeclaration()) {

            reportError("Invalid parameter list: missing or malformed parameter declaration");

            return false;

        }

        // Parse additional parameter declarations separated by commas

        while (match(",")) {

            // Move to the next token after the comma

            // Parse the next parameter declaration

            if (!parseParameterDeclaration()) {

                reportError("Invalid parameter list: missing or malformed parameter declaration after comma");

                return false;

            }

        }

        return true;

    }

    private boolean parseForStatement() {

        Node forStatementNode = new Node("for\_statement");

        // Check if the current token is '('

        if (!match("(")) {

            reportError("Expected '(' after 'for'");

            return false;

        }

        // Parse initialization expression

        Node initializationExpressionNode = new Node("initialization\_expression");

        if (!parseExpression()) {

            reportError("Invalid initialization expression in for statement");

            return false;

        }

        currentNode.addChild(initializationExpressionNode); // Add initialization expression to the current node

        // Check if the current token is ';'

        if (!match(";")) {

            reportError("Expected ';' after initialization expression");

            return false;

        }

        // Parse condition expression

        Node conditionExpressionNode = new Node("condition\_expression");

        if (!parseExpression()) {

            reportError("Invalid condition expression in for statement");

            return false;

        }

        currentNode.addChild(conditionExpressionNode); // Add condition expression to the current node

        // Check if the current token is ';'

        if (!match(";")) {

            reportError("Expected ';' after condition expression");

            return false;

        }

        // Parse update expression

        Node updateExpressionNode = new Node("update\_expression");

        if (!parseExpression()) {

            reportError("Invalid update expression in for statement");

            return false;

        }

        currentNode.addChild(updateExpressionNode); // Add update expression to the current node

        // Check if the current token is ')'

        if (!match(")")) {

            reportError("Expected ')' after update expression");

            return false;

        }

        // Parse loop body

        if (!parseBlock()) {

            reportError("Invalid block in for statement");

            return false;

        }

        currentNode.addChild(forStatementNode); // Add for statement node to the current node

        return true;

    }

    private boolean parseBlock() {

        Node blockNode = new Node("block");

        // Check if the current token is '{'

        if (!match("{")) {

            reportError("Expected '{' to start a block");

            return false;

        }

        // Parse statements inside the block until '}'

        while (!tokens.get(currentTokenIndex).getValue().equals("}")) {

            if (!parseStatement()) {

                reportError("Invalid statement inside block");

                return false;

            }

        }

        // Check if the current token is '}'

        if (!match("}")) {

            reportError("Expected '}' to end a block");

            return false;

        }

        currentNode.addChild(blockNode); // Add block node to the current node

        return true;

    }

    private boolean parseWhileStatement() {

        if (!match("(")) {

            reportError("Expected '(' after 'while'");

            return false;

        }

        // Create a node for the while statement

        Node whileStatementNode = new Node("while\_statement");

        // Add the while statement node as a child of the current node

        currentNode.addChild(whileStatementNode);

        // Move the current node pointer to the while statement node

        currentNode = whileStatementNode;

        // Parse condition expression

        Node conditionExpressionNode = new Node("condition\_expression");

        if (!parseExpression()) {

            reportError("Invalid condition expression in while statement");

            return false;

        }

        // Add the condition expression node as a child of the while statement node

        whileStatementNode.addChild(conditionExpressionNode);

        if (!match(")")) {

            reportError("Expected ')' after condition expression");

            return false;

        }

        // Parse loop body

        if (!parseBlock()) {

            return false;

        }

        // Restore the current node pointer to the parent node

        currentNode = currentNode.getParent();

        return true;

    }

    private boolean parseDoWhileStatement() {

        // Create a node for the do-while statement

        Node doWhileStatementNode = new Node("do\_while\_statement");

        // Add the do-while statement node as a child of the current node

        currentNode.addChild(doWhileStatementNode);

        // Move the current node pointer to the do-while statement node

        currentNode = doWhileStatementNode;

        // Parse loop body

        if (!parseBlock()) {

            reportError("Invalid block in do-while statement");

            return false;

        }

        if (!match("while")) {

            reportError("Expected 'while' after block in do-while statement");

            return false;

        }

        if (!match("(")) {

            reportError("Expected '(' after 'while'");

            return false;

        }

        // Parse condition expression

        Node conditionExpressionNode = new Node("condition\_expression");

        if (!parseExpression()) {

            reportError("Invalid condition expression in do-while statement");

            return false;

        }

        // Add the condition expression node as a child of the do-while statement node

        doWhileStatementNode.addChild(conditionExpressionNode);

        if (!match(")")) {

            reportError("Expected ')' after condition expression");

            return false;

        }

        if (!match(";")) {

            reportError("Expected ';' after do-while statement");

            return false;

        }

        // Restore the current node pointer to the parent node

        currentNode = currentNode.getParent();

        return true;

    }

    private boolean match(String keyword) {

        if (currentTokenIndex < tokens.size() && tokens.get(currentTokenIndex).getValue().equals(keyword)) {

            currentTokenIndex++;

            return true;

        }

        return false;

    }

    private boolean parsePostfixExpression() {

        Node postfixExpressionNode = new Node("postfix\_expression");

        if (match("IDENTIFIER")) {

            // Identifier

            Node identifierNode = new Node(tokens.get(currentTokenIndex)); // Create a node for the identifier

            postfixExpressionNode.addChild(identifierNode); // Add identifier node to the postfix expression node

        } else if (isConstant(tokens.get(currentTokenIndex))) {

            // Constant

            Node constantNode = new Node(tokens.get(currentTokenIndex)); // Create a node for the constant

            postfixExpressionNode.addChild(constantNode); // Add constant node to the postfix expression node

            currentTokenIndex++;

        } else if (match("STRING")) {

            // String literal

            Node stringLiteralNode = new Node(tokens.get(currentTokenIndex)); // Create a node for the string literal

            postfixExpressionNode.addChild(stringLiteralNode); // Add string literal node to the postfix expression node

        } else if (match("(")) {

            // Parenthesized expression

            if (!parseExpression()) {

                reportError("Invalid expression inside parentheses");

                return false;

            }

            if (!match(")")) {

                reportError("Expected ')' after expression");

                return false;

            }

        } else {

            // Invalid postfix expression

            reportError("Invalid postfix expression");

            return false;

        }

        // Check for postfix operators (e.g., array access, function call, increment, decrement)

        while (true) {

            if (match("[")) {

                // Array access

                Node arrayAccessNode = new Node(tokens.get(currentTokenIndex)); // Create a node for array access

                postfixExpressionNode.addChild(arrayAccessNode); // Add array access node to the postfix expression node

                if (!parseExpression()) {

                    reportError("Invalid expression inside array access");

                    return false;

                }

                if (!match("]")) {

                    reportError("Expected ']' after array index");

                    return false;

                }

            } else if (match("(")) {

                // Function call

                Node functionCallNode = new Node(tokens.get(currentTokenIndex)); // Create a node for function call

                postfixExpressionNode.addChild(functionCallNode); // Add function call node to the postfix expression node

                if (!parseArgumentExpressionList(functionCallNode)) {

                    // Error handling if the argument expression list parsing fails

                    return false;

                }

                if (!match(")")) {

                    reportError("Expected ')' after argument list");

                    return false;

                }

            } else if (match("++") || match("--")) {

                // Increment or decrement

                Node incrementDecrementNode = new Node(tokens.get(currentTokenIndex)); // Create a node for increment/decrement

                postfixExpressionNode.addChild(incrementDecrementNode); // Add increment/decrement node to the postfix expression node

            } else {

                // No more postfix operators

                break;

            }

        }

        currentNode.addChild(postfixExpressionNode); // Add postfix expression node to the current node

        return true;

    }

    private boolean parseArgumentExpressionList(Node functionCallNode) {

        // Create a node for the argument expression list

        Node argumentExpressionListNode = new Node("argument\_expression\_list");

        // Add the argument expression list node as a child of the function call node

        functionCallNode.addChild(argumentExpressionListNode);

        // Parse the first argument expression, if present

        if (!parseAssignmentExpression()) {

            // Error handling if the assignment expression parsing fails

            reportError("Invalid argument expression in function call");

            return false;

        }

        // Parse additional argument expressions separated by commas

        while (match(",")) {

            // Move to the next token after the comma

            // Parse the next argument expression

            if (!parseAssignmentExpression()) {

                // Error handling if the assignment expression parsing fails

                reportError("Invalid argument expression in function call");

                return false;

            }

        }

        return true;

    }

    private boolean match(TokenType expectedTokenType) {

        if (currentTokenIndex < tokens.size() && tokens.get(currentTokenIndex).getType() == expectedTokenType) {

            currentTokenIndex++;

            return true;

        }

        return false;

    }

    private void reportError(String message) {

        System.out.println("Error: " + message + tokens.get(currentTokenIndex) + currentTokenIndex);

    }

    private boolean match(TokenType expectedTokenType, Node parentNode) {

        if (currentTokenIndex < tokens.size() && tokens.get(currentTokenIndex).getType() == expectedTokenType) {

            Node tokenNode = new Node(tokens.get(currentTokenIndex).getValue()); // Create a node for the token

            parentNode.addChild(tokenNode); // Add the token node to the parent node

            currentTokenIndex++;

            return true;

        }

        return false;

    }

    private boolean match(String expectedTokenType, Node parentNode) {

        if (currentTokenIndex < tokens.size() && tokens.get(currentTokenIndex).getValue().equals(expectedTokenType) ) {

            Node tokenNode = new Node(tokens.get(currentTokenIndex).getValue()); // Create a node for the token

            parentNode.addChild(tokenNode); // Add the token node to the parent node

            currentTokenIndex++;

            return true;

        }

        return false;

    }

    private boolean is\_a\_type(String type){

        for( String str : dataTypes ){

            if( str.equals(type) )

                return true;

        }

        return false;

    }

    private boolean isConstant(Token token) {

        TokenType type = token.getType();

        return type == TokenType.DECIMAL ||

                type == TokenType.OCTAL ||

                type == TokenType.BINARY ||

                type == TokenType.HEX ||

                type == TokenType.FLOAT ||

                type == TokenType.STRING ||

                type == TokenType.Character;

    }

}

## Code description :

**Data Types**: The dataTypes array holds the various data types that the parser can recognize.

**Constructor**: The Parser constructor initializes the tokens list, sets the current token index to 0, and creates the root node of the syntax tree.

**Parse Method**: The parse method initiates the parsing process by calling the parse Program method.

**Parse Program**: The parse Program method iterates over all tokens and attempts to parse function declarations and declarations until no more tokens are left.

**Parse Assignment Expression**: This method parses assignment expressions, which include operations like assignment (=), addition assignment (+=), subtraction assignment (-=), and so on.

**Parse Conditional Expression**: This method parses conditional expressions, which are expressions involving the ternary operator (? :).

**Parse Logical Expressions**: The methods parseLogicalOrExpression and parseLogicalAndExpression parse logical OR (||) and logical AND (&&) expressions, respectively.

**Parse Inclusive Or Expression**: This method parses inclusive OR expressions, which are expressions involving the OR operator (|).

**Parse Exclusive Or Expression**: This method parses exclusive OR expressions, which are expressions involving the XOR operator (^).

**Parse And Expression**: This method parses AND expressions, which are expressions involving the AND operator (&).

**Parse Equality Expression**: This method parses equality expressions, which are expressions involving the equality (==) and inequality (!=) operators.

**Parse Shift Expression**: This method parses shift expressions, which include operations like left shift (<<) and right shift (>>).

**Parse Additive Expression**: This method parses additive expressions, which include operations like addition (+) and subtraction (-).

**Parse Multiplicative Expression**: This method parses multiplicative expressions, which include operations like multiplication (\*), division (/), and modulus (%).

**Parse Cast Expression**: This method parses cast expressions, which are expressions that convert one data type to another.

**Parse Type Name**: This method parses type names, which are used in cast expressions and sizeof expressions.

**Parse Unary Expression**: This method parses unary expressions, which are expressions involving unary operators like unary plus (+), unary minus (-), logical negation (!), and bitwise complement (~).

**Parse Relational Expression**: This method parses relational expressions, which are expressions involving relational operators like less than (<), greater than (>), less than or equal to (<=), and greater than or equal to (>=).

**Parse Labeled Statement**: This method parses labeled statements, which are statements that are preceded by a label. This includes identifiers followed by a colon (:), case statements, and default statements.

**Parse Statement**: This method parses statements, which are the smallest standalone elements of an imperative programming language

**Parse Iteration Statement  :**method handles the parsing of iteration statements, which include while, do-while, and for loops. For each type of loop, it checks if the current token matches the keyword (while, do, for), and if it does, it creates a new node for that loop type and adds it to the current node. Then it parses the components of the loop (such as the condition expression and the body of the loop) and adds them as children to the loop node. If any part of the loop cannot be parsed correctly, it reports an error and returns false.

**Parse Jump Statement :**method is responsible for parsing jump statements, which include goto, continue, break, and return statements. For each type of jump statement, it checks if the current token matches the keyword (goto, continue, break, return), and if it does, it creates a new node for that jump statement type and adds it to the current node. Then it parses any additional components of the jump statement (such as the identifier for a goto statement or the expression for a return statement) and adds them as children to the jump statement node. If any part of the jump statement cannot be parsed correctly, it reports an error and returns false.

The peek method is used to look at the next token without consuming it. This is useful in situations where you need to know what the next token is in order to decide how to parse the current token, but you don’t want to actually consume the next token yet.

**Parse Selection Statement:** method handles the parsing of selection statements, which include if and else if statements. It checks if the current token matches the keyword (if), and if it does, it creates a new node for the if statement and adds it to the current node. Then it parses the components of the if statement (such as the condition expression and the body of the if statement) and adds them as children to the if statement node. If any part of the if statement cannot be parsed correctly, it reports an error and returns false.

**Parse Expression Statement:** method is responsible for parsing expression statements. It calls the parse Expression method to parse the expression, and if the expression is successfully parsed, it consumes an optional semicolon and returns true. If the expression cannot be parsed, it reports an error and returns false.

**parse Expression** :method parses an assignment expression and then checks if the current token is a comma. If it is, it continues to parse assignment expressions until there are no more commas. If the assignment expression cannot be parsed, it returns false.

**parse Declaration:** method is responsible for parsing declarations. It first parses the type specifier, and if that is successful, it checks if the current token is an identifier. If it is, it consumes the identifier and then parses the declarator list and a semicolon. If any part of the declaration cannot be parsed correctly, it reports an error and returns false.

**Parse Declarator List** :method parses a list of declarators. It first parses a declarator, and if that is successful, it checks if the current token is a comma. If it is, it continues to parse declarators until there are no more commas. If the declarator cannot be parsed, it returns false.

**Parse Declarator:**method is responsible for parsing declarators. It first checks if the current token is an asterisk, indicating a pointer declarator. If it is, it consumes the asterisk and then checks if the current token is an identifier. If it is, it consumes the identifier and then checks if the current token is an equals sign. If it is, it parses the initializer. If any part of the declarator cannot be parsed correctly, it reports an error and returns false.

**Parse Initializer:** method parses an assignment expression and adds it as a child to the initializer node. If the assignment expression cannot be parsed, it reports an error and returns false.

**Parse Compound Statement:** method is responsible for parsing compound statements, which are enclosed in braces {}. It first checks if the current token is an opening brace {. If it is, it consumes the brace and then parses declarations and statements until it encounters a closing brace }. If any part of the compound statement cannot be parsed correctly, it reports an error and returns false.

**Parse Pointer :**method parses pointers. It first checks if the current token is an asterisk \*, indicating a pointer. If it is, it consumes the asterisk and then parses any type qualifiers. After that, it parses an optional pointer. If the current token is not an asterisk, it returns false.

**Parse Direct Declarator:**method is responsible for parsing direct declarators. It first checks if the current token is an identifier. If it is, it consumes the identifier and returns true. If the current token is not an identifier, it checks if it is an opening parenthesis (. If it is, it consumes the parenthesis and then parses a declarator. If the declarator is successfully parsed, it checks if the current token is a closing parenthesis ). If it is, it consumes the parenthesis and returns true. If the current token is not a closing parenthesis, it reports an error and returns false.

**Parse Constant Expression** :method parses constant expressions. It first checks if the current token represents a constant. If it does, it creates a node for the constant and adds it to the constant expression node. Then it moves to the next token and adds the constant expression node to the current node. If the current token does not represent a constant, it reports an error and returns false.

**Parse Identifier List:** method is incomplete in the provided code snippet. It is expected to parse a list of identifiers, but the implementation is not provided. The method should iterate over the tokens, checking if each token is an identifier, and add each identifier to the identifier list node. If a token is not an identifier, it should report an error and return false. If all identifiers are successfully parsed, it should return true. The method should also handle commas , separating the identifiers in the list. If

# Test cases of all project :