Lexical Analysis & Parsing Project

AIN SHAMS UNIVERSITY FACULTY OF ENGINEERING 1-CREDIT HOURS ENGINEERING PROGRAMS COMPUTER ENGINEERING AND SOFTWARE SYSTEMS PROGRAM

2023-2024

Lexical Analysis & Parsing Project

DESIGN OF COMPILERS

CSE 439

Code & Demo Link: Click Here

Name	ID
Ahmed Nezar Ahmed Hussien	21P0025
AbdulRahman Hesham Kamel Seleim Abduallah	21P0153
Omar Alaa Eldin Fareed Elnahass	21P0197
Kirollos Ehab Magdy Halim	21P0006
Tsneam Ahmed Eliwa Zky Mohamed Ghoname	21P0284

Course Coordinator:

Dr. Wafaa Samy & Eng. Ahmed Salama

Contents

1 Introduction

1.1 Lexical Analysis Definition

Lexical Analysis,also known as Scanner ,is the initial phase of a compiler.Its primary purpose is to transform a high-level input program into a sequence of tokens.Lexical Analysis can be implemented using Deterministic Finite Automata.
Input: Stream of characters.
Output: The output is a sequence of tokens that is sent to the parser for syntax analysis.

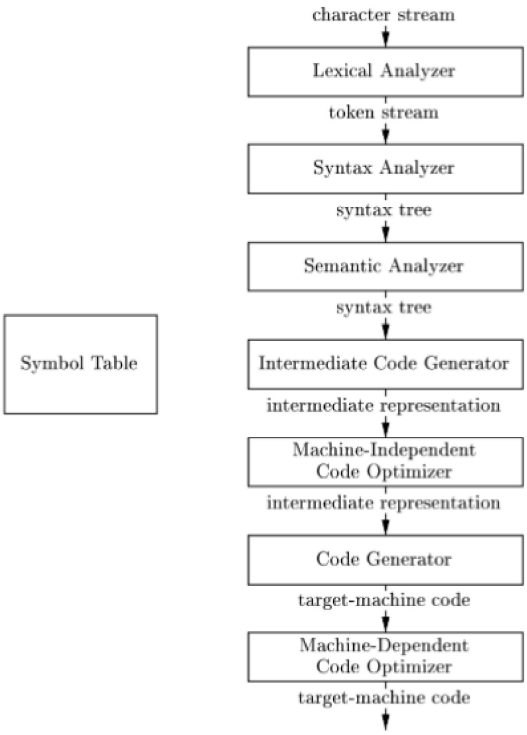


Figure 1: Phases Of compilers

1.2 Tokens

A lexical token is a sequence of characters that can be treated as a unit in the grammar of the programming languages.
Examples of tokens include:

- Type tokens (e.g., identifiers, numbers, real values).
- Punctuation tokens (e.g., keywords like “if,” “void,” “return”).
- Alphabetic tokens (e.g., keywords such as “for,” “while,” “if”).

Example of Non-Tokens: Comments, preprocessor directive, macros, blanks, tabs, newline, etc.

1.3 Lexemes:

A lexeme refers to the sequence of characters matched by a pattern to form a corresponding token. In other words, it’s the input characters that make up a single token.

Examples of lexemes include:

- “float”
- “abs_zero_Kelvin”
- “=”
- “-”
- “273”
- “;”

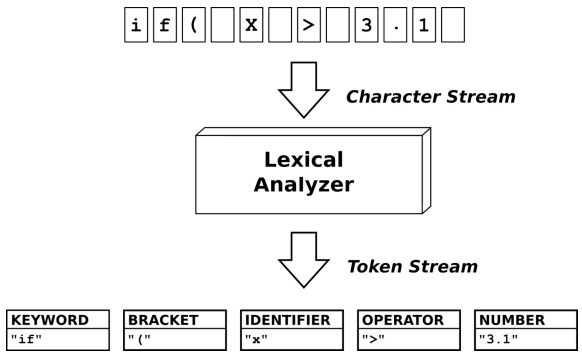


Figure 2: Lexical Analysis Process

1.4 How Lexical Analysis Works?

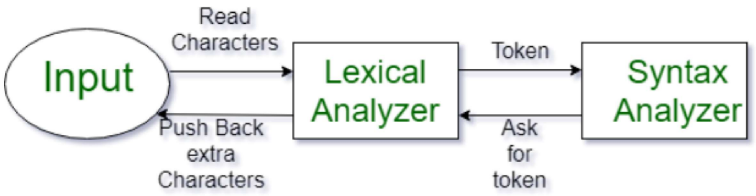


Figure 3: Lexical Analysis

1. **Input Preprocessing:** Clean up the input text by removing comments, whitespace, and non-essential characters.
2. **Tokenization:** Break the input text into a sequence of tokens by matching characters against predefined patterns or regular expressions.
3. **Token Classification:** Determine the type of each token (e.g., keywords, identifiers, operators).
4. **Token Validation:** Check that each token adheres to the language rules (e.g., valid variable names, correct syntax for operators).
5. **Output Generation:** Produce a list of tokens, which can be passed to subsequent compilation stages.

Lexeme	token
while	while
(	lparen
y	identifier
>=	Comparison
t	identifier
)	Rparen
y	identifier
=	Assignment
y	identifier
-	Arithmetic
3	integer
;	Finish of a statement

Figure 4: Lexemes and Tokens

example, consider the program

```
int main()
{
  // 2 variables
  int a, b;
  a = 10;
 return 0;
}
```

All the valid tokens are:

```
'int'  'main'  '('  ')'  '{'  'int'  'a' ','  'b'  ';'
 'a'  '='  '10'  ';' 'return'  '0'  ';'  '}'
```

Figure 5: Valid Tokens Example

1.5 Error Detection in lexical Analysis:

The lexical analyzer detects errors based on the grammar rules of the programming language (such as C). When an error occurs, it reports the specific row and column numbers where the issue was found.

1.6 Advantages of Lexical Analysis:

1. **Efficient Parsing:** Lexical analysis tokenizes the input program, making subsequent parsing and semantic analysis more streamlined. By breaking down the code into smaller units (tokens), it simplifies the overall compilation process.

2. **Early Error Detection:** Lexical analysis identifies lexical errors (such as misspelled keywords or undefined symbols) at an early stage in the compilation process. This proactive error checking enhances the overall efficiency of the compiler or interpreter by catching issues sooner.

3. **Enhanced Efficiency:** Once the source code is tokenized, subsequent compilation or interpretation phases operate more efficiently. Parsing and semantic analysis benefit from working with this streamlined tokenized input.

1.7 Disadvantages of Lexical Analysis:

1. **Limited Context:**Lexical analysis operates based on individual tokens and does not take into account the overall context of the code. This limitation can sometimes lead to ambiguity or misinterpretation of the code's intended meaning, especially in languages with complex syntax or semantics.

2. **Overhead:**While lexical analysis is necessary for the compilation process, it introduces an additional layer of overhead. Tokenizing the source code requires extra computational resources, which can impact the overall performance of the compiler.

3. **Debugging Challenges:**Lexical errors detected during the analysis phase may not always provide clear indications of their origins in the original source code. Debugging such errors can be challenging, especially if they result from subtle mistakes in the lexical analysis process.

2 Regular Expressions

Definition

Regular expressions (regex) are patterns used for matching character combinations in strings. They are widely used in text processing and are supported by many programming languages and tools. Here's a basic explanation of some common components of regular expressions:

Components of Regular Expressions:

1. **Literals:** Literals are characters that match themselves. For example, the regular expression `hello` matches the string "hello" exactly.

2. **Metacharacters:** Metacharacters are special characters with predefined meanings in regular expressions. Some common metacharacters include:

   - `.` (dot): Matches any single character except newline.
   - `|` (pipe): Alternation operator, matches either the expression before or after it.
   - `*`: Matches zero or more occurrences of the preceding character or group.
   - `+`: Matches one or more occurrences of the preceding character or group.
   - `?`: Matches zero or one occurrence of the preceding character or group.
   - `^`: Anchors the match to the beginning of the string.
   - `$`: Anchors the match to the end of the string.
   - \: Escape character, allows using metacharacters as literals.
   - `[ ]`: Character class, matches any single character within the brackets.
   - `^[ ]`: Negated character class, matches any single character not within the brackets.

3. **Quantifiers:** Quantifiers specify the number of occurrences of the preceding character or group that should be matched. Some common quantifiers include:

   - `*`: Matches zero or more occurrences.
   - `+`: Matches one or more occurrences.
   - `?`: Matches zero or one occurrence.
   - {`n`}: Matches exactly `n` occurrences.
   - {`n`,}: Matches `n` or more occurrences.
   - {`n`,`m`}: Matches between `n` and `m` occurrences (inclusive).

4. **Grouping:** Parentheses (`( )`) are used to group characters or subexpressions together. They are used to apply quantifiers and other operators to multiple characters.

5. **Anchors:** Anchors are used to specify positions in the string where matches should occur. Common anchors include:
    - `^`: Matches the beginning of a line.
    - `$`: Matches the end of a line.
    - `\b`: Matches a word boundary.
6. **Escape Sequences:** Escape sequences allow including special characters as literals. For example, `\.` matches a literal period instead of the special meaning of `.`.

How Regular Expressions Work:

When you apply a regular expression to a string, the regex engine searches for matches by:

- Starting at the beginning of the string (or as specified by anchors).
- Attempting to match the pattern specified by the regular expression.
- Moving along the string one character at a time, trying to match the pattern.
- Backtracking when necessary if a match is not found.
- Returning all matches found or performing the specified action (e.g., replacement).

Operator	**Description**
*	Zero or more occurrences of the preceding element
?	Zero or one occurrence of the preceding element
+	One or more occurrences of the preceding element
\	Escape character for special characters
.	Any single character except newline
()	Grouping for capturing subexpressions
	Character class, matches any character within the brackets
—	Alternation, matches either the expression before or after
{}	Specifies the exact number of occurrences or a range
^	Anchors the match to the beginning of the line
$	Anchors the match to the end of the line

Table 1: Regular Expression Operators

Implemented Regular Expressions

```
String num_regex = "\\d+(\\.\\d+)?((e|E)(['+']|-)?\\d+)?";
String binary_octal_hex_regex = "(((0b|0B)[0-1]+)|(0[0-7]+)|((0x|0X)[0-9a-fA-F]+))";
String bad_binary_octal_hex_regex = "(0b|0B)[0-9]+|0[0-9]+|(0x|0X)[0-9a-zA-Z]+";
String num_dataType_regex = "(ULL|LL|L|UL|F|ull|ll|l|ul|f)?";
String bad_Identifiers = "\\d+[a-zA-Z_#$@]+\\d*|[a-zA-Z_#$@]+[#$@]+[a-zA-Z_#$@\\d]+";
Pattern tokenPattern = Pattern.compile(
        "\"(?:[^\"\\\\]|\\\\.)*\"|" +             // Match double-quoted strings
        bad_binary_octal_hex_regex + "|" +  // Match binary, octal, decimal, and hexadecimal numbers
        binary_octal_hex_regex + "|" +  // Match binary, octal, decimal, and hexadecimal numbers
        "'.'|" +                    // Match single characters within single quotes
        "\"|'|" +                   // Match bad puncatuation
        bad_Identifiers + "|" +     // Match bad variable names
        num_regex + num_dataType_regex + "|" +           // Match numbers
        "\\b[a-zA-Z_][a-zA-Z0-9_]*\\b(\\()|" + // Match identifiers (Function)
        "\\}(?:\\s*\\w+)?\\s*;|" + // Match End Block of Struct
        "^(struct|typedef( )struct)\\s+(\\w+)\\s*[\\{|;]|" + // Match identifiers (Struct)
        "\\b[a-zA-Z_][a-zA-Z0-9_]*\\b((?!\\()|(?!\\{))|" + // Match identifiers (Variable)
        "\\(|\\)|\\{|\\}|;|,|\\[|\\]|" +    // Match parentheses, braces, semicolon, comma
        "\\+\\+|--|=|≠|≤|≥|&&|" + // Match comparison and logical operators
        "\\-\\>|\\+\\=|\\-\\=|\\*\\=|\\/\\=|\\%\\=|\\&\\=|\\|\\=|\\|\\||" + // Match compound assignment operators
        "\\+|\\*|/|%|\\^\\=|\\<\\<\\=|\\>\\>\\=|\\<\\◁\\>\\>|<▷|" + // Match arithmetic and bitwise operators
        "\\-|\\+|\\*|\\=|\\&\\&|\\|\\||\\!|\\&|\\||\\^|\\~|\\?|" + // Match miscellaneous operators
        "\\:|\\+\\+|\\-\\-|\\." // Match other operators
);
```

Figure 6: Regex Code

1. `\\d+(\\.\\d+)?((e|E)(['+']|-)?\\d+)?`
    - This regular expression matches numeric literals with optional decimal points and exponent notation.
    - `\\d+`: Matches one or more digits.
    - `(\\.\\d+)?`: Matches an optional decimal point followed by one or more digits.
    - `((e|E)(['+']|-)?\\d+)?`: Matches an optional exponent part, including 'e' or 'E', followed by an optional sign ('+' or '-'), and one or more digits.
2. `(((0b|0B)[0-1]++)|(0[0-7]+)|((0x|0X)[0-9a-fA-F]+))`
    - This regular expression matches binary, octal, and hexadecimal literals.
    - It consists of three main alternatives:
        - `(0b|0B)[0-1]++`: Matches binary literals prefixed with '0b' or '0B', followed by one or more binary digits.
        - `0[0-7]+`: Matches octal literals prefixed with '0', followed by one or more octal digits.
        - `(0x|0X)[0-9a-fA-F]+`: Matches hexadecimal literals prefixed with '0x' or '0X', followed by one or more hexadecimal digits.

3. `(0b|0B)[0-9]+|0[0-9]+|(0x|0X)[0-9a-zA-Z]+`

   - This regular expression matches bad binary, octal, and hexadecimal values in a string.
   - `(0b`0B`)[0-9]+`:
     - `(0b`0B`)`—: This is a capturing group that matches either "0b" or "0B", indicating the prefix for binary literals, regardless of case sensitivity.
     - `[0-9]+`: Matches one or more digits. However, in binary literals, only digits 0 and 1 are allowed, but this regex allows any digit.
     - This part of the regex matches bad binary values.
   - `0[0-9]+`:
     - Matches bad octal values.
     - `0`: Matches the prefix "0" indicating an octal literal.
     - `[0-9]+`: Matches one or more digits. However, in octal literals, only digits 0-7 are allowed, but this regex allows any digit after the leading zero.
   - `(0x`0X`)[0-9a-zA-Z]+`:
     - This part matches bad hexadecimal values.
     - `(0x`0X`)`—: This capturing group matches either "0x" or "0X", indicating the prefix for hexadecimal literals, regardless of case sensitivity.
     - `[0-9a-zA-Z]+`: Matches one or more characters that can be digits (0-9) or letters (a-z, A-Z). However, in hexadecimal literals, only digits 0-9 and letters a-f or A-F are allowed, but this regex allows any alphanumeric character.

4. `(ULL|LL|L|UL|F|ull|ll|l|ul|f)?`

   - This regular expression matches optional C/C++ data type suffixes for numeric literals.
   - It allows for various combinations of uppercase and lowercase letters representing data types, followed by an optional occurrence of those letters.

5. `\\d+[a-zA-Z_#$@]+\\d*|[a-zA-Z_#$@]+[#$@]+[a-zA-Z_#$@\\d]+`

   - This regular expression identifies strings that likely represent invalid or unconventional identifiers. It matches two distinct cases:

     Case 1:

     - Begins with one or more digits (‘`\d+`‘)
     - Followed by one or more letters, underscores, #, $, or symbols (‘`[a-zA-Z_#$@]+`‘)
     - May optionally end with zero or more digits (‘`\d*`‘)

     Case 2:

     - Begins with one or more letters, underscores, #, $, or symbols (‘`[a-zA-Z_#$@]+`‘)
     - Must include one or more #, $, or symbols immediately after (‘`[#$@]+`‘)
     - Followed by one or more letters, underscores, #, $, symbols, or digits (‘`[a-zA-Z_#$@\d]+`‘)
   - **Examples of matches:**
     - 123abc
     - abc123
     - _hello#
     - my_Var#1
     - hello#$world
     - var##$123
   - **Purpose:** This regular expression is likely intended for programming language syntax validation. Typical identifier conventions discourage starting with numbers or the excessive use of special characters like #, $, and .

6. `\"(?:[^\"\\\\]|\\\\.)*\"`
   - Matches double-quoted strings.
   - Allows for any character except a double quote or a backslash inside the quotes unless the backslash is escaping another character.
7. `'.'`
   - Matches a single character enclosed in single quotes.
8. `\"|'`
   - Matches a single double quote or single quote, representing bad punctuation when not closed.
9. `\\b[a-zA-Z_][a-zA-Z0-9_]*\\b(\\()`
   - Matches identifiers that are immediately followed by an opening parenthesis '(', suggesting function names.
   - Begins with a letter or underscore, followed by any number of letters, digits, or underscores.
10. `\}(?:\s*\w+)?\s*;`
    - Matches the end of a block for structures, possibly followed by a word (identifier) and then a semicolon.
11. `^(struct|typedef( )struct)\\s+(\\w+)\\s*[\\{|;]`
    - Matches the declaration of a structure, starting at the beginning of a line.
    - Followed by an optional 'typedef', then the structure name, and possibly a curly brace or semicolon.
12. `\\b[a-zA-Z_][a-zA-Z0-9_]*\\b((?!\\()|(?!\\{))`
    - Matches identifiers that are not immediately followed by an opening parenthesis '(' or a curly brace '', suggesting variable names.
13. `\\(|\\)|\\{|\\}|;|,`
    - Matches parentheses, braces, semicolons, and commas.
14. `\\+\\+|--|==|!=|<=|>=|&&`
    - Matches various comparison and logical operators:
      - `++`: Increment operator.
      - `--`: Decrement operator.
      - `==`: Equality comparison operator.
      - `!=`: Inequality comparison operator.
      - `<=`: Less than or equal to comparison operator.
      - `>=`: Greater than or equal to comparison operator.
      - `&&`: Logical AND operator.
15. `\\-\\>|\\+\\=|\\-\\=|\\*\\=|\\/\\=|\\%\\=|\\&\\=|\\|\\=|\\|\\|`
    - Matches compound assignment operators:
      - `->`: Structure pointer access operator.
      - `+=`: Addition assignment operator.
      - `-=`: Subtraction assignment operator.

- `*=`: Multiplication assignment operator.
- `/=`: Division assignment operator.
- `%=`: Modulus assignment operator.
- `&=`: Bitwise AND assignment operator.
- `||=`: Logical OR assignment operator.

16. `\\+|\\*|/|%|\\^\=|\\<\\<\\=|\\>\\>\\=|\\<\\<|\\>\\>|<|>`

- Matches arithmetic and bitwise operators:
  - `+`: Addition operator.
  - `*`: Multiplication operator.
  - `/`: Division operator.
  - `%`: Modulus operator.
  - `^=`: Bitwise XOR assignment operator.
  - `<<=`: Left shift assignment operator.
  - `>>=`: Right shift assignment operator.
  - `<<`: Left shift operator.
  - `>>`: Right shift operator.
  - `<`: Less than operator.
  - `>`: Greater than operator.

17. `\\-|\\+|\\*|\\=|\\&\\&|\\|\\||\\!|\\&|\\||\\^|\\~|\\?`

- Matches various miscellaneous operators:
  - `-`: Subtraction operator.
  - `+`: Addition operator.
  - `*`: Multiplication operator.
  - `=`: Assignment operator.
  - `&&`: Logical AND operator.
  - `||`: Logical OR operator.
  - `!`: Logical NOT operator.
  - `&`: Bitwise AND operator.
  - `|`: Bitwise OR operator.
  - `^`: Bitwise XOR operator.
  - `~`: Bitwise complement operator.
  - `?`: Ternary conditional operator.

18. `\\:|\\+\\+|\\-\\-|\\.`

- Matches colons, increment, decrement, and the dot operator:
  - `::` Colon operator.
  - `++`: Increment operator.
  - `--`: Decrement operator.
  - `.`: Dot operator.

3 Code Explanation

Our Compiler implementation works under the work of 5 classes, 3 classes have the logic & 2 classes handles GUI.

3.1 C File Reader Class

3.1.1 Initialization

The `cReader` class is initialized with two instance variables: `path` to hold the file path of the .c file, and `cLines` to store the lines of the .c file.

```
private String path;
private ArrayList<String> cLines = new ArrayList<String>();
```

3.1.2 Reading and Cleaning Lines

The `getClines()` method reads each line from the specified .c file, removes comments (both single-line and multi-line) and preprocessors, and stores the cleaned lines in the `cLines` ArrayList.

```
public ArrayList<String> getClines() {
    // Specify the path to your .c file
    String filePath = path;

    try (BufferedReader br = new BufferedReader(new FileReader(filePath))) {
        String line;

        // Read file line by line
        while ((line = br.readLine()) != null) {
            // Process each line as needed
            cLines.add(line);
        }
    } catch (IOException e) {
        // Handle any IO exceptions
        e.printStackTrace();
    }
    cleanClines();
    return cLines;
}
```

3.1.3 Cleaning Process

- Single-line comments (`//`) are detected and removed entirely from each line.
- Multi-line comments (`/* ... */`) are identified and replaced with an empty string.
- Preprocessor directives (`#`) are detected and removed from the line.

3.1.4 Handling Block Comments

A boolean flag (`blockComment`) is used to handle multi-line comments. It is set when encountering `/*` and unset when encountering `*/`. Lines within block comments are replaced with an empty string until the end of the comment block is reached.

```
private void cleanClines(){
    boolean blockComment = false;
    for (int i = 0; i < cLines.size(); i++) {
        String line = cLines.get(i);
        if (blockComment) {
            if (line.matches("(.)*\\*/")) {
                blockComment = false;
            }
            cLines.set(i, "");
        } else {
            if (line.matches("(.)*//(.)*")){
                cLines.set(i, cLines.get(i).replaceAll("//(.)*", ""));
            }
            else if (line.matches("(.)*/\\*(.)*\\*/")){
                cLines.set(i, cLines.get(i).replaceAll("/\\*(.)*\\*/", ""));
            }
            else if (line.matches("(.)*/\\*(.)*")) {
                blockComment = true;
                cLines.set(i, cLines.get(i).replaceAll("/\\*(.)*", ""));
            }
        }
        if (line.matches("(.)*#(.)*")){
            cLines.set(i, cLines.get(i).replaceAll("#(.)*", ""));
        }
    }
}
```

3.1.5 Providing Access and Clearing

The `cReader` class offers methods to access the cleaned lines (`getClines()`) and the original lines (`cLinesGetter()`). It also provides a method (`clear()`) to clear the stored lines from the `cLines` ArrayList.

```
public ArrayList<String> cLinesGetter() {
    return cLines;
}

public void clear(){
    cLines.clear();
}
```

3.1.6 Usage

Users can instantiate the `cReader` class with or without a file path, then call the `getClines()` method to read and prepossess the lines of the specified .c file. After prepossessing, the cleaned lines can be accessed for further processing or analysis.

3.2 Lexer Class

3.2.1 Brief Overview

The `Lexer` class is responsible for tokenizing C code files. Tokenization involves breaking down the source code into smaller units called tokens, which represent the basic building blocks of the code.

3.2.2 Initialization

The `Lexer` class initializes a `cReader` object to read the C code file. It also defines predefined tokens for operators, punctuation, and keywords. We create 3 Array Lists to store Predefined operators, predefined & predefined keywords.

```
public class Lexer {
    private static Dictionary<String, ArrayList<String>> predefinedTokens = new Hashtable<>();
    public cReader reader;
    private ArrayList<Token> tk = new ArrayList<>();

    public Lexer(String path){
        reader = new cReader(path);
        ArrayList<String> cOperators = new ArrayList<>();
        ArrayList<String> cPunctuation = new ArrayList<>();
        ArrayList<String> cKeywords = new ArrayList<>();
                // Add C-language operators to the ArrayList
        cOperators.add("+");cOperators.add("-");cOperators.add("*");
        cOperators.add("/");cOperators.add("%");
        cOperators.add("=");cOperators.add("==");cOperators.add("!=");
        cOperators.add(">");cOperators.add("<");
        cOperators.add(">=");cOperators.add("<=");cOperators.add("&&");cOperators.add("||");
        cOperators.add("!");
        cOperators.add("&");cOperators.add("|");
        cOperators.add("^");cOperators.add("~");
        cOperators.add("<<");
        cOperators.add(">>");cOperators.add(">>>");
        cOperators.add("?:");cOperators.add("++");
        cOperators.add("--");
        cOperators.add("+=");cOperators.add("-=");cOperators.add("*=");
        cOperators.add("/=");cOperators.add("%=");
        cOperators.add("&=");cOperators.add("|=");
        cOperators.add("^=");cOperators.add("<<=");
        cOperators.add(">>=");
        cOperators.add(">>>=");cOperators.add("->");cOperators.add(".");
        cOperators.add("?");cOperators.add(":");

        // Add C-language punctuations to the ArrayList
        cPunctuation.add("(");cPunctuation.add(")");
        cPunctuation.add("{");cPunctuation.add("}");
        cPunctuation.add("]");cPunctuation.add(";");
        cPunctuation.add(",");cPunctuation.add("[");
        cPunctuation.add("'");cPunctuation.add("\"");

        // Add C-language keywords to the ArrayList
        cKeywords.add("auto");cKeywords.add("break");
        cKeywords.add("case");cKeywords.add("char");
        cKeywords.add("const");cKeywords.add("continue");
        cKeywords.add("default");cKeywords.add("do");
        cKeywords.add("double");cKeywords.add("else");cKeywords.add("enum");
```

```
        cKeywords.add("extern");
        cKeywords.add("float");cKeywords.add("for");cKeywords.add("goto");
        cKeywords.add("if");
        cKeywords.add("int");cKeywords.add("long");cKeywords.add("register");
        cKeywords.add("return");
        cKeywords.add("short");cKeywords.add("signed");cKeywords.add("sizeof");
        cKeywords.add("static");
        cKeywords.add("struct");cKeywords.add("switch");cKeywords.add("typedef");
        cKeywords.add("union");
        cKeywords.add("unsigned");cKeywords.add("void");cKeywords.add("volatile");
        cKeywords.add("while");
        cKeywords.add("inline");cKeywords.add("restrict");

        predefinedTokens.put("Operators", cOperators);
        predefinedTokens.put("Punctuations", cPunctuation);
        predefinedTokens.put("Keywords", cKeywords);
    }
}
```

3.2.3 Tokenization

The `tokenize()` method reads each line of the C code, matches tokens using regular expressions (Explained in the earlier section), and adds them to the token list. It works by traversing on line by line in the file then start extracting tokens from each lexeme. It compares each token with pattern that was defined by regular expressions & accordingly divides it into tokens.

```
public void tokenize() {
    tk.clear();
    Token.idCount = 1;
    ArrayList<String> id_struct = new ArrayList<>();
    ArrayList<String> lines = reader.getClines();
    boolean inBlockComment = false;
    String num_regex = "\\d+(\\.\\d+)?((e|E)(['+']|-)?\\d+)?";
    String binary_octal_hex_regex = "(((0b|0B)[0-1]+)|(0[0-7]+)|((0x|0X)[0-9a-fA-F]+))";
    String bad_binary_octal_hex_regex = "(0b|0B)[0-9]+|0[0-9]+|(0x|0X)[0-9a-zA-Z]+";
    String num_dataType_regex = "(ULL|LL|L|UL|F|ull|ll|l|ul|f)?";
    String bad_Identifiers = "\\d+[a-zA-Z_#$@]+\\d*|[a-zA-Z_#$@]+[#$@]+[a-zA-Z_#$@\\d]+";
    Pattern tokenPattern = Pattern.compile(
            "\"(?:[^\"\\\\]|\\\\.)*\"|" +              // Match double-quoted strings
            bad_binary_octal_hex_regex + "|" +  // Match binary, octal, decimal, and hexadecimal numb
            binary_octal_hex_regex + "|" +  // Match binary, octal, decimal, and hexadecimal numbers
            "'.'|" +                    // Match single characters within single quotes
            "\"|'|" +                   // Match bad puncatuation
            bad_Identifiers + "|" +     // Match bad variable names
            num_regex + num_dataType_regex + "|" +           // Match numbers
            "\\b[a-zA-Z_][a-zA-Z0-9_]*\\b(\\()|" + // Match identifiers (Function)
            "\\}(?:\\s*\\w+)?\\s*;|" + // Match End Block of Struct
            "^(struct|typedef( )struct)\\s+(\\w+)\\s*[\\{|;]|" + // Match identifiers (Struct)
            "\\b[a-zA-Z_][a-zA-Z0-9_]*\\b((?!\\()|(?!\\{))|" + // Match identifiers (Variable)
            "\\(|\\)|\\{|\\}|;|,|\\[|\\]|" +    // Match parentheses, braces, semicolon, comma
            "\\+\\+|--|==|!=|<=|>=|&&|" + // Match comparison and logical operators
            "\\-\\>|\\+\\=|\\-\\=|\\*\\=|\\/\\=|\\%\\=|\\&\\=|\\|\\=|\\|\\||" + // Match compound ass
            "\\+|\\*|/|%|\\^\\=|\\<\\<\\=|\\>\\>\\=|\\<\\<|\\>\\>|<|>|" + // Match arithmetic and bit
            "\\-|\\+|\\*|\\=|\\&\\&|\\|\\||\\!|\\&|\\||\\^|\\~|\\?|" + // Match miscellaneous operato
            "\\:|\\+\\+|\\-\\-|\\." // Match other operators
    );
```

```
31      boolean isStruct = false;
32      boolean structDataType;
33      int lineNum = 1;
34      for (String line : lines) {
35          structDataType = false;
36          // Skip lines starting with "#" and comments
37          if (!line.trim().startsWith("#")) {
38              // Match tokens using regular expression
39              Matcher matcher = tokenPattern.matcher(line);
40              while (matcher.find()) {
41                  String token = matcher.group();
42                  if (token.matches("\\}(?:\\s*\\w+)?\\s*;") && isStruct){
43                      isStruct = false;
44                      String filteredStr = token.replaceAll("\\s", "").replaceAll(";", "")
45                              .replaceAll("}", "");
46                      tk.add(new Token("Puncatuations", "}", lineNum));
47                      tk.add(new Token("Puncatuations", ";", lineNum));
48                      if (!filteredStr.isEmpty()){
49                          id_struct.add(filteredStr);
50                          tk.add(new Token("Identifiers (Struct)", filteredStr, lineNum));
51                      }
52                      continue;
53                  }
54                  if (isPunctuation(token)) {
55                      tk.add(new Token("Puncatuations", token, lineNum));
56                      // Disable Struct Data Type Flag
57                      if (token.equals(";"))
58                          structDataType = false;
59                  } else if (isOperator(token)) {
60                      tk.add(new Token("Operators", token, lineNum));
61                  } else if (token.matches("(0b|0B)[0-9]+") && !token.replaceAll("[2-9]+", "").equals(t
62                          || token.matches("0[0-9]+") && !token.replaceAll("[8-9]+", "").equals(token)
63                          || token.matches("(0x|0X)[0-9a-zA-Z]+") && !token.replaceFirst("(0x|0X)", ""
64                          .replaceAll("[g-zG-Z]+", "").equals(token.replaceFirst("(0x|0X)", ""))) {
65                      tk.add(new Token("Bad Integers", token, lineNum));
66                  } else if (token.matches(binary_octal_hex_regex + "|" + num_regex + num_dataType_rege
67                      if (token.matches(num_regex + "(ul|UL|l|L)")) {
68                          tk.add(new Token("Long", token, lineNum));
69                      } else if (token.matches(num_regex + "(ull|ULL|ll|LL)")) {
70                          tk.add(new Token("Long Long", token, lineNum));
71                      }
72                      else if (token.matches(binary_octal_hex_regex) ||
73                              token.matches("(0|([1-9][0-9]*))")){
74                          tk.add(new Token("Integers", token, lineNum));
75                      } else{
76                          tk.add(new Token("Floats", token, lineNum));
77                      }
78                  } else if (token.startsWith("'") && token.endsWith("'")) {
79                      tk.add(new Token("Characters", token, lineNum));
80                  } else if (token.startsWith("\"") && token.endsWith("\"")) {
81                      tk.add(new Token("Strings", token, lineNum));
82                  } else if (isKeyword(token)) {
83                      if (token.equals("struct")) {
84                          isStruct = true;
85                      }
86                      tk.add(new Token("Keywords", token, lineNum));
```

```
87                  } else {
88                      if (id_struct.contains(token) && !structDataType){
89                          id_struct.add(token);
90                          tk.add(new Token("Identifiers (Struct)", token, lineNum));
91                          structDataType = true;
92                      }
93                      // Check if the token is a function or a variable
94                      else if (isFunction(token)) {
95                          token = token.replaceFirst("\\(", "");
96                          tk.add(new Token("Puncatuations", "(", lineNum));
97                          if (isKeyword(token)) {
98                              tk.add(new Token("Keywords", token, lineNum));
99                          } else {
100                             tk.add(new Token("Identifiers (Function)", token, lineNum));
101                         }
102                     } else if (isStruct(token)) {
103                         isStruct = true;
104                         token = token.replaceFirst("typedef", "");
105                         tk.add(new Token("Keywords", "typedef", lineNum));
106
107                         token = token.replaceFirst("struct", "");
108                         tk.add(new Token("Keywords", "struct", lineNum));
109
110                         token = token.replaceAll("\\s", "");
111                         while(!token.equals(token.replaceFirst("\\{", ""))){
112                             token = token.replaceFirst("\\{", "");
113                             tk.add(new Token("Puncatuations", "{", lineNum));
114                         }
115                         while(!token.equals(token.replaceFirst(";", ""))){
116                             token = token.replaceFirst(";", "");
117                             tk.add(new Token("Puncatuations", ";", lineNum));
118                         }
119                         id_struct.add(token);
120                         tk.add(new Token("Identifiers (Struct)", token, lineNum));
121
122                     } else {
123                         if (token.contains("}")) {
124                             token = token.replaceFirst("}", "");
125                             tk.add(new Token("Puncatuations", "}", lineNum));
126                         }
127                         if (token.contains(";")) {
128                             token = token.replaceFirst(";", "");
129                             tk.add(new Token("Puncatuations", ";", lineNum));
130                         }
131                         if (token.matches(bad_Identifiers)){
132                             tk.add(new Token("Bad Identifiers", token, lineNum));
133                         } else if (!token.isEmpty()) {
134                             tk.add(new Token("Identifiers (Variable)", token, lineNum));
135                         }
136                     }
137                 }
138             }
139         }
140         lineNum++;
141     }
142 }
```

3.2.4 Token Categories

Here we implemented some methods to check for the following while reading the C file:

- Function
- Struct
- Variable
- Operator
- Punctuation
- Keyword

The logic works by comparing & checking with predefined tokens(dictionary) or by checking if it matches the pattern of regex

```
 private boolean isFunction(String token) {
        return token.contains("(");
    }

    private boolean isStruct(String token) {
        return !token.replaceFirst("struct", "").equals(token);
    }

    private boolean isVariable(String token) {
        // Check if the token is followed by an equal sign or is a standalone identifier
        return token.matches("[a-zA-Z_][a-zA-Z0-9_]*\\s*(=\\s*.*|;)");
    }

    private boolean isOperator(String token) {
        return predefinedTokens.get("Operators").contains(token);
    }

    private boolean isPunctuation(String token) {
        return predefinedTokens.get("Punctuations").contains(token);
    }

    private boolean isKeyword(String token) {
        return predefinedTokens.get("Keywords").contains(token);
    }
```

3.3 Token Class

3.3.1 Brief Overview

The `Token` class represents individual tokens identified during the tokenization process performed by the `Lexer` class. It encapsulates information such as the name of the token, its attribute value, and the line number where it appears in the source code. Thus it is used in constructing **symbol table**.

3.3.2 Constructor

The constructor of the `Token` class initializes its attributes based on the parameters provided: `name`, `attrVal`, and `lineNum`.

```
public class Token {
    String name;
```

```
3       String attrVal;
4       Integer lineNum;
5
6       // Static variables for identifier management
7       static Integer idCount = 1;
8       Integer entryNum = 0;
9       static Integer numObj = 0;
10      String identifier = "";
11      String identifierType = "";
12
13      public Token(String name, String attrVal, Integer lineNum) {
14          this.name = name;
15          this.attrVal = attrVal;
16          this.lineNum = lineNum;
17          if (name.contains("Identifiers") && !name.contains("Bad")){
18              this.identifier = "id" + idCount;
19              this.entryNum = idCount;
20              idCount++;
21              identifierType = name.replaceFirst("Identifiers \\(", "")
22                      .replaceFirst("\\)", "");
23      }
24  }
```

3.3.3 Identifier Management

The `Token` class manages identifiers by assigning unique identifier names (`identifier`) and entry numbers (`entryNum`) to each identifier token encountered during tokenization. Then after assigning them we will use it in the symbol table

4 GUI Application

The "C Lexer & Parser" is a software application with a graphical user interface designed to analyze C language source code. It is a tool for analyzing C code, offering features to tokenize, create symbol tables, check syntax, and visualize code structure with parse trees.

1. Upload File

This button is likely for uploading C source code files that you want to process with this tool.

2. Tokenize

Clicking this button would typically trigger the lexer part of the tool to convert the source code into a series of tokens. Tokenization is the process of breaking up the input text into a sequence of tokens, which are the meaningful elements of the language like keywords, identifiers, symbols, etc.

3. Symbol Table

This might display a table containing all the symbols (identifiers, constants, functions, etc.) used in the C source code with additional information such as their types, scope, and perhaps memory addresses.

4. Parse

This would initiate the parsing process, where the sequence of tokens generated by the lexer is analyzed to make sure it follows the grammatical structure of the language. Parsing checks the syntax and builds a representation of the code that can be used for further processing, like constructing a parse tree or an abstract syntax tree (AST).

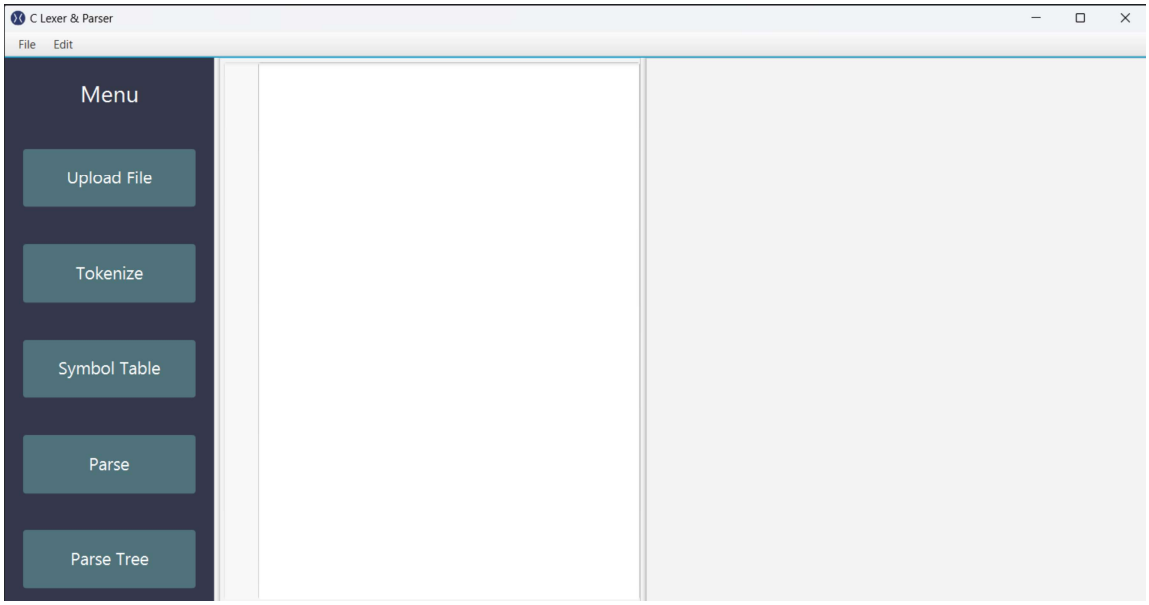


Figure 7: Main Window

5. Parse Tree

After parsing, this button might display the parse tree (or syntax tree), which is a tree representation of the syntactic structure of the source code based on the grammar of the language. This tree illustrates the hierarchy of language constructs and their relationship to each other within the code.

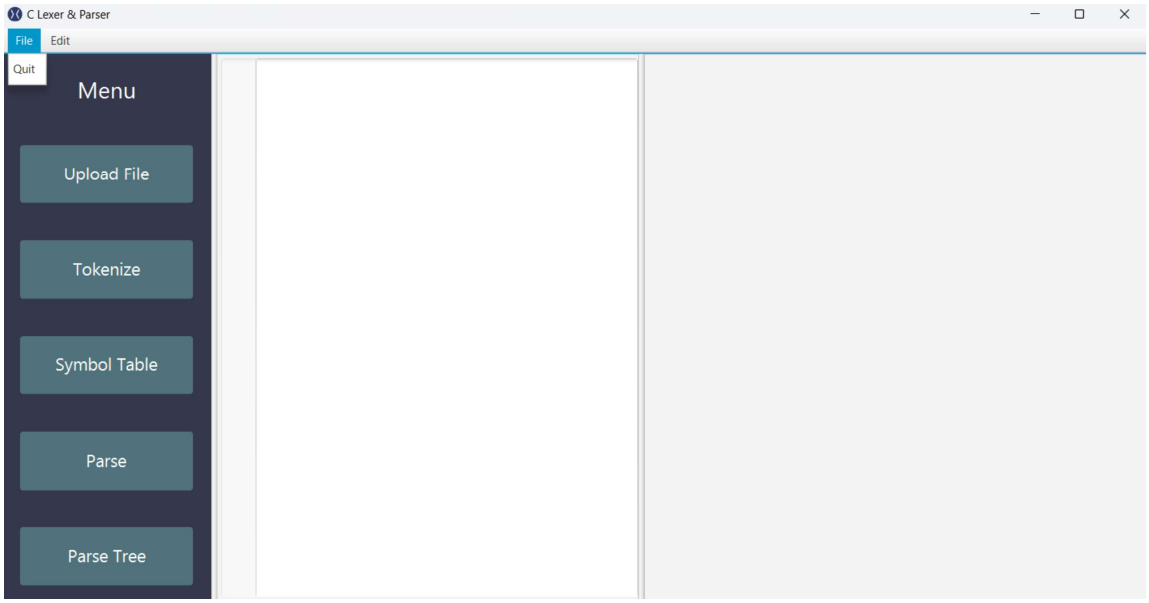


Figure 8: File Button

White Area

The large white area to the right seems to be the main workspace where the output of each action (token list, symbol table, parse tree) might be displayed after the user performs an operation using the buttons in the menu. The top-left menu options labeled "File" and "Edit" are typical of application GUIs and would likely contain further options to save, load, or modify files, among other functions.

File Button

In the image, the "File" menu in the upper left corner is shown with one visible option: "Quit." This option is standard in many software applications and is used to close the application. When you select "Quit," it typically prompts the application to end its processes and close the window, effectively exiting the program. Depending on the application's design, you might be prompted to save any unsaved work before the application closes to prevent loss of data.

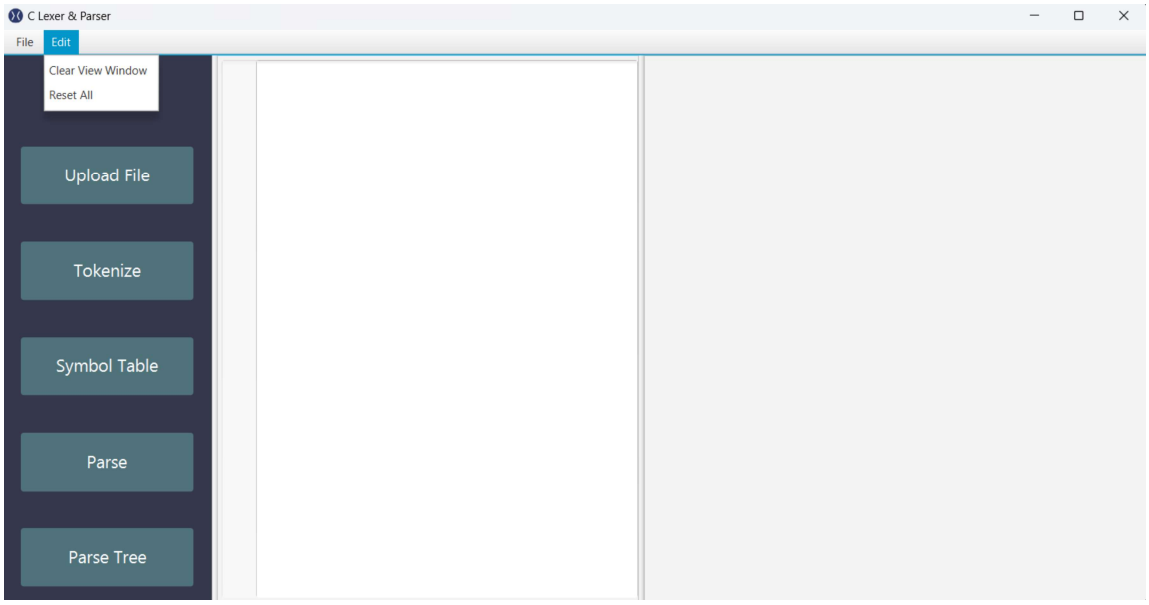


Figure 9: Edit Button

Edit Button

In the upper left corner of the GUI, there's a menu labeled "Edit." This menu is typically used to house functions that allow the user to modify or manipulate the current document or files they are working on. In many applications, an "Edit" menu might include options like undo/redo, cut, copy, paste, find, and replace. However, in the context of this specific "C Lexer & Parser" application, the "Edit" menu contents are not visible, so we can only speculate that it would contain functions relevant to editing the code or data that the user is working with. It may include options to modify the input C code or adjust how the lexing and parsing operations are performed.

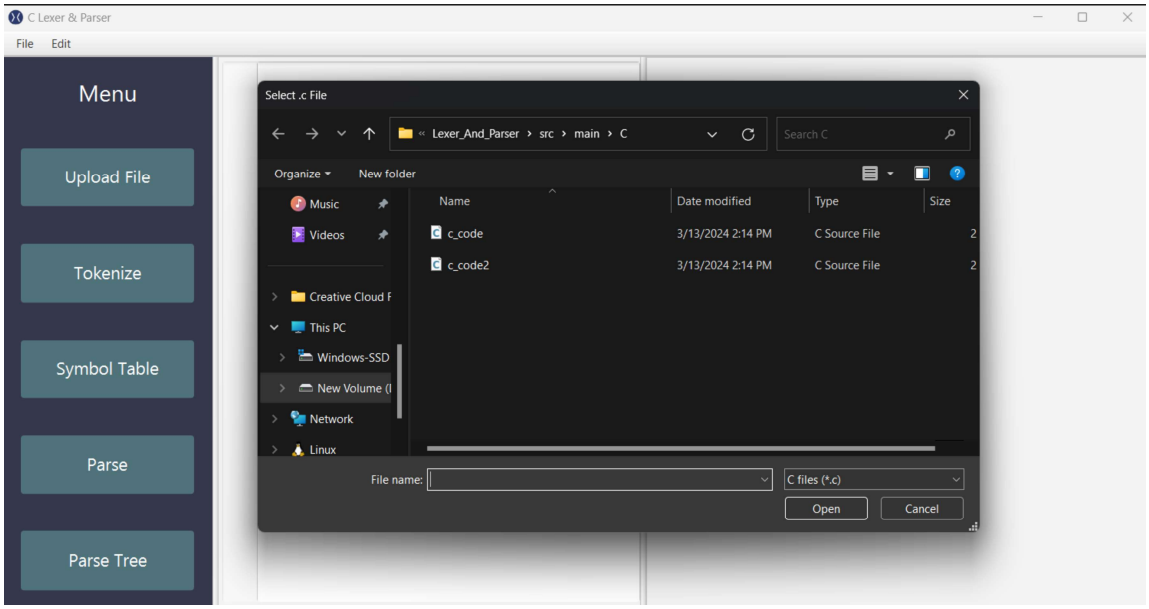


Figure 10: Upload File

The GUI shown in the image is a file dialog box, which is part of the "C Lexer & Parser" software application. This dialog box has been opened by clicking the "Upload File" button in the application's main window.

- **Dialog Box Title**: "Select .c File" indicates the purpose of the dialog is to select a C language source file.
- **Breadcrumb Navigation**: At the top, there's a path (Lexer_And_Parser ¿ src ¿ main ¿ C), showing the current directory, which allows the user to navigate the file system hierarchy.
- **Back and Forward Buttons**: These can be used to navigate to previously visited directories.
- **Main Area**: Displays files and folders from the current directory. In this case, two C source files named "c_code" and "c_code2" are visible, along with details such as modification date, type, and size.
- **Search Bar**: It allows the user to search for files within the current directory.
- **File Name Field**: Here, the user can type the name of a file they wish to open.
- **File Type Dropdown**: It allows the user to filter the files shown in the dialog box. It's currently set to "C files (*.c)" which will only show files with the .c extension.
- **Open and Cancel Buttons**: "Open" would confirm the selection and presumably load the file into the "C Lexer & Parser" application for processing. "Cancel" would close the dialog box without making a selection.

This dialog box is a common component in GUI applications, facilitating file opening and uploading operations.

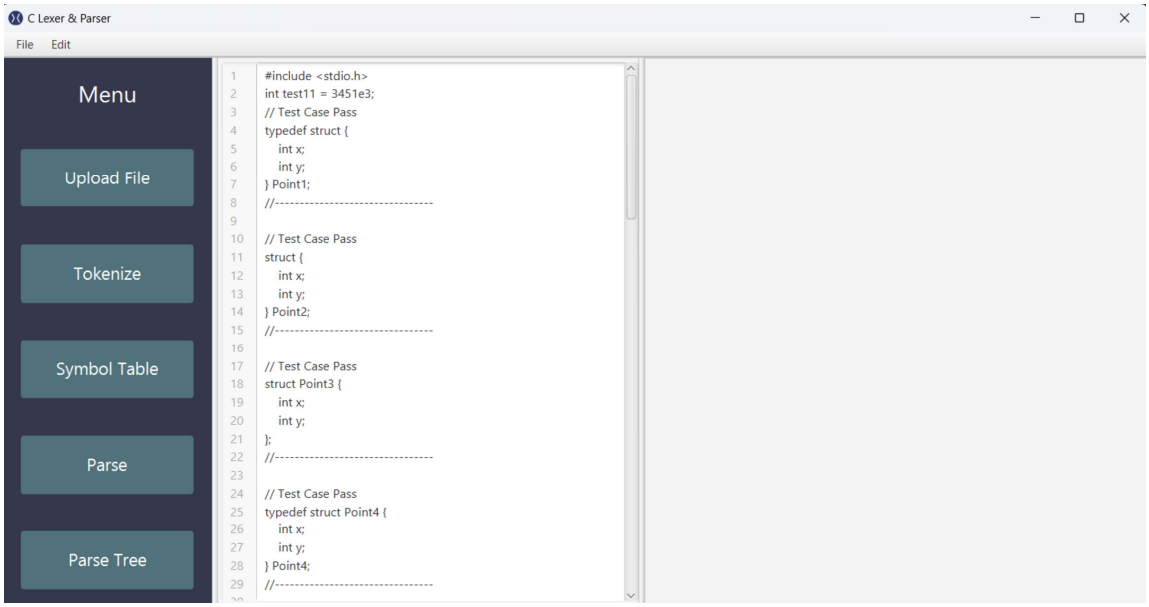


Figure 11: C File Code View

The GUI shown in the image is a menu bar from the software application "C Lexer Parser". The menu bar provides options for interacting with the application and potentially managing C source code files.

- **File Menu:** This menu likely offers functionalities related to file management, such as opening existing C source files, saving the currently edited file, or creating new files.
- **Edit Menu:** This menu might provide options for editing the C code within the application, such as copy, paste, undo, and redo functionalities.
- **Menu:** The presence of a menu named simply "Menu" suggests it might be a placeholder or a work-in-progress feature. Its functionalities are unclear from this screenshot.
- **Tokenize Menu:** This option likely initiates the process of tokenizing the loaded C source code file. Tokenization is the process of breaking the code down into smaller meaningful units (tokens) for further analysis by the application.
- **Parse Menu:** This option likely triggers the parsing stage, where the tokenized code is analyzed according to the grammar rules of the C language. This allows the application to understand the structure and meaning of the code.
- **Parse Tree Menu:** This option might be used to display a visual representation of the parsed code structure, often referred to as a parse tree. This can be helpful for debugging and understanding the code's syntax.
- **Symbol Table Menu:** This option likely provides access to the symbol table, a data structure that stores information about variables and other symbols used within the C code.

The "C Lexer Parser" software seems to be designed for lexical and syntactical analysis of C code. The menu bar offers functionalities for loading, manipulating, and analyzing C source code files.

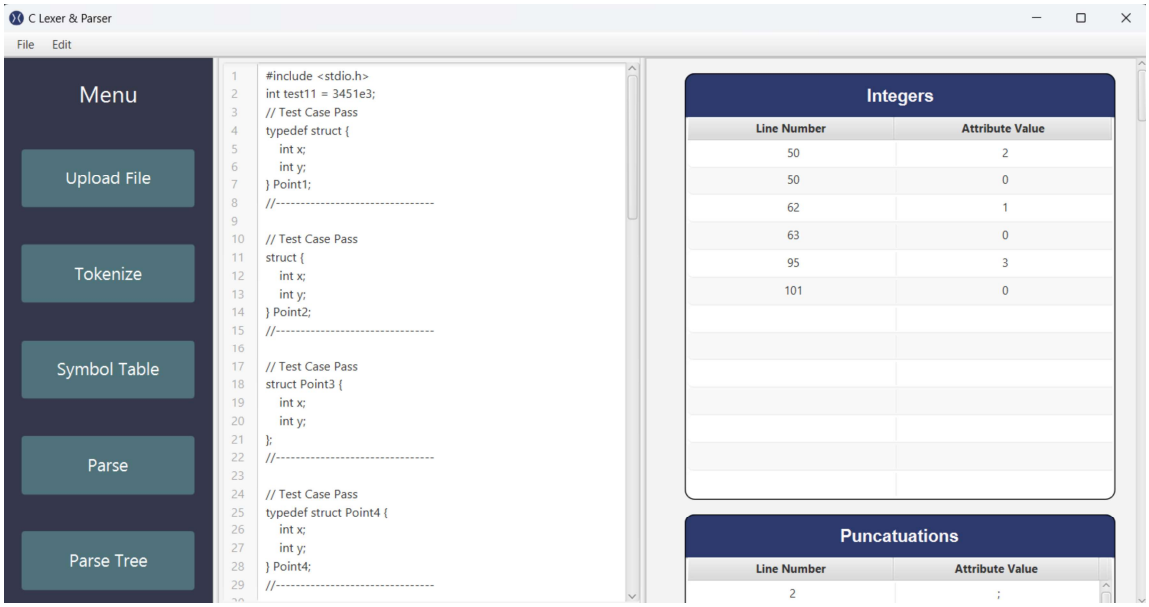
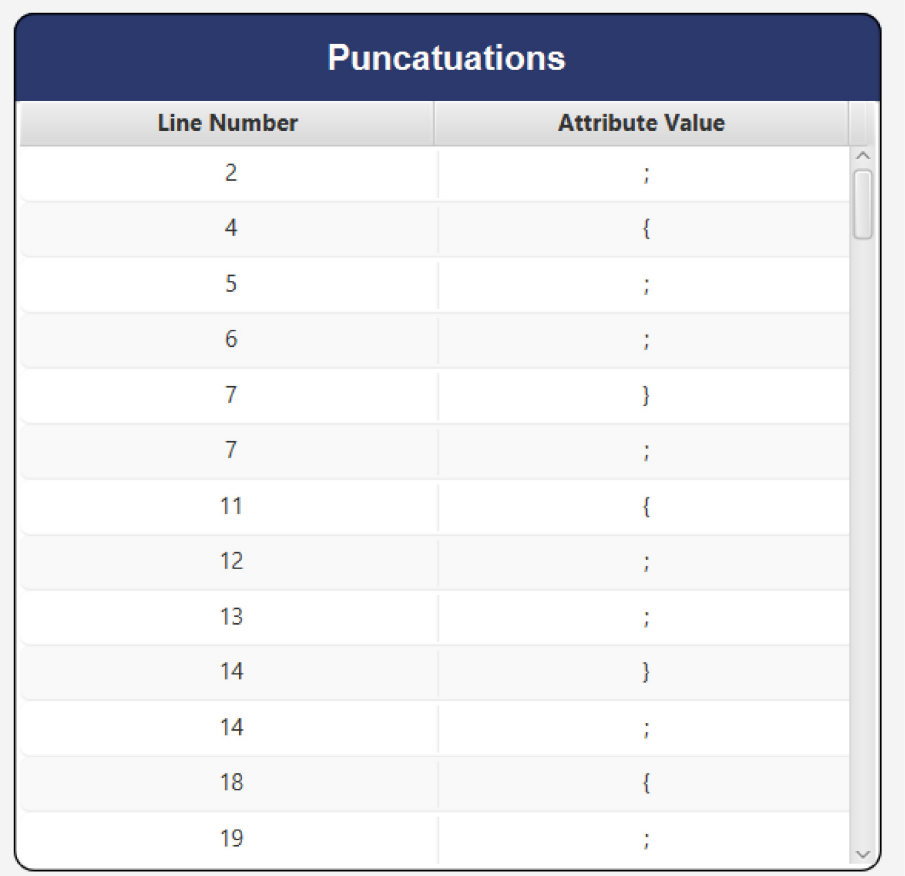


Figure 12: Tokenize Button

The screenshot displays a tokenization results window within the "C Lexer & Parser" software application. This window presents the results of tokenizing a C source code file, providing a structured view for analysis.

- **Table Header:** The row at the top likely indicates the column names within the tokenization results table. Typical headers could be "Token Type", "Token Value", "Line Number", and potentially others.
- **Tokenized Data:** The subsequent rows contain the actual tokenized data. Each row likely represents a single token identified within the C code during the tokenization process.
- **Token Types:** This column likely displays the type of token. Examples would include keywords (like 'int', 'if', 'while'), identifiers (variable names), operators, and punctuation symbols.
- **Token Values:** This column displays the actual value of the token within the source code. For example, a variable name would be listed here.
- **Line Numbers:** This column might indicate the line number in the original source code where the corresponding token was found.
- **Navigation/Search (Possibly):** The window might include navigation buttons or a search bar to allow users to quickly find specific tokens within the table.

The "C Lexer & Parser" software appears to feature a dedicated window for inspecting the results of tokenization. This table-based format provides a structured representation of the C code, likely used for further analysis or debugging within the application.



Puncatuations

Line Number	Attribute Value
2	;
4	{
5	;
6	;
7	}
7	;
11	{
12	;
13	;
14	}
14	;
18	{
19	;

Figure 13: Punctuations Tokens Table

Punctuation marks are essential for structuring code and separating different elements within the program's syntax. The lexer identifies punctuation tokens to aid in understanding the structural aspects of the code.

- **Table Headers**: "Punctuation Symbol", "Line Number", (potentially) "Column Number"
- **Punctuation Symbols**: This column lists the various punctuation symbols found in the code, such as periods (.), commas (,), semicolons (;), colons (:), braces (), etc.
- **Line Numbers**: Indicates the line where each punctuation symbol appears in the source code.
- **Column Numbers (Optional)**: Provides the column position of the punctuation symbol within its line.

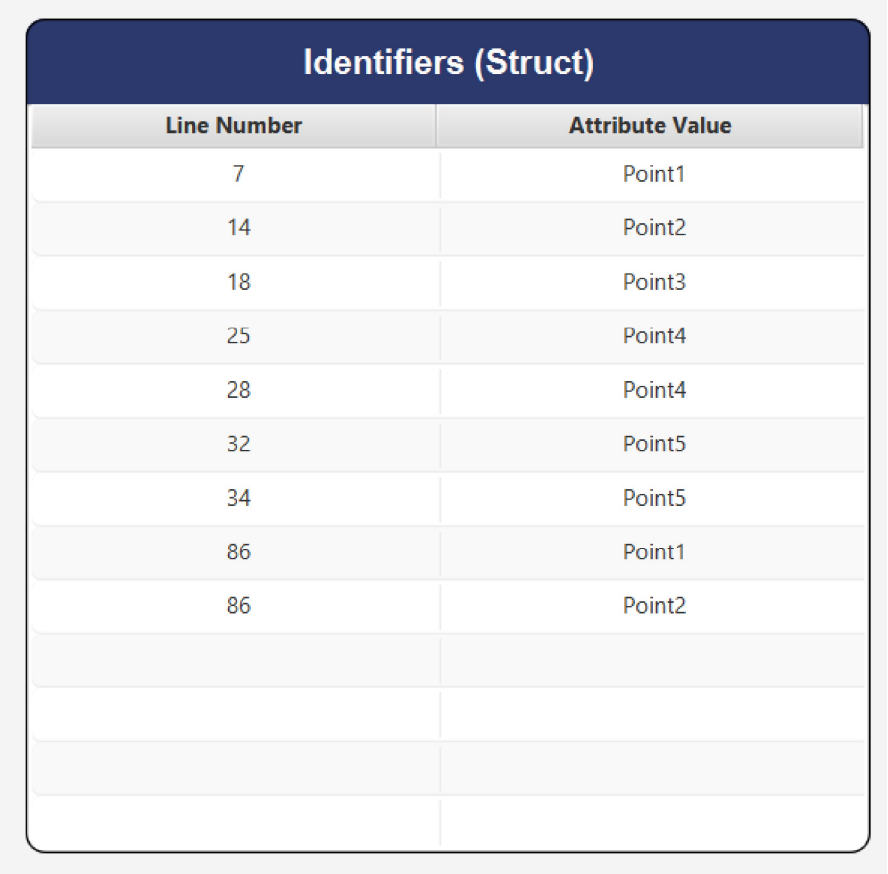
Operators

Line Number	Attribute Value
2	=
42	+
50	%
50	==
54	/
58	*
62	=
63	=
63	<
63	++
64	*=
73	=
74	=

Figure 14: Operators Tokens Table

Operators are crucial for performing calculations, comparisons, logical operations, and assignments within the code. Analyzing operator tokens is vital for understanding the code's behavior.

- **Table Headers**: "Token Type", "Token Value", "Line Number", (potentially) "Column Number"
- **Token Types**: Classifications like arithmetic, comparison, logical, assignment, etc.
- **Token Values**: The actual operator symbols (+, -, ¡, , etc.).
- **Line and Column Numbers**: Location of the operator within the source code.



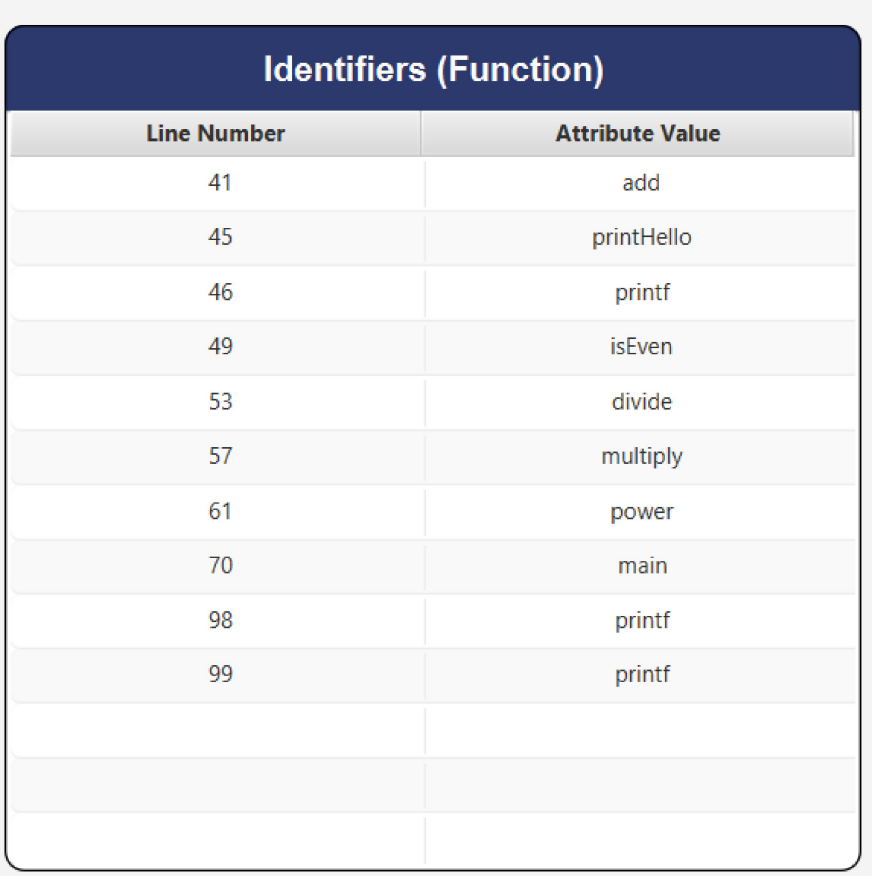
Identifiers (Struct)

Line Number	Attribute Value
7	Point1
14	Point2
18	Point3
25	Point4
28	Point4
32	Point5
34	Point5
86	Point1
86	Point2

Figure 15: Identifiers (Struct) Tokens Table

The Identifiers (Struct) Tokens Table lists the structure name identifiers detected by the lexer within the source code. Structures are user-defined data types that group variables of different types under a single name. This table likely provides details about what structures are defined, where they appear in the code, and the members contained within these structures.

- **Table Headers**: "Struct Name", "Line Number", (potentially) "Column Number", (potentially) "Member Variables"
- **Struct Names**: The names used to define structures in the code.
- **Line and Column Numbers**: The location of the struct definition.
- **Member Variables (Optional)**: A list of variable names and types declared within the struct.



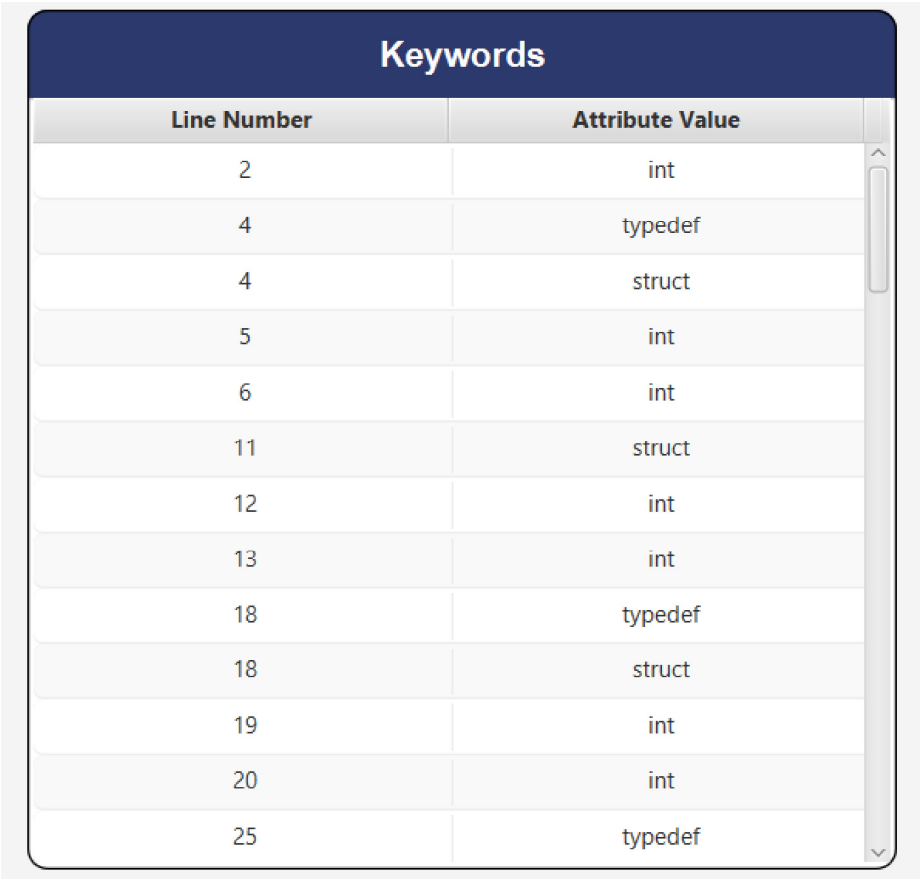
Identifiers (Function)

Line Number	Attribute Value
41	add
45	printHello
46	printf
49	isEven
53	divide
57	multiply
61	power
70	main
98	printf
99	printf

Figure 16: Identifiers (Function) Tokens Table

The Identifiers (Function) Tokens Table lists function names detected within the source code. Functions are blocks of code designed to perform specific tasks. This table provides information about the defined functions, helping understand where they are declared and used in the code.

- **Table Headers**: "Function Name", "Line Number", (potentially) "Column Number", (potentially) "Parameter Types"
- **Function Names**: The names used to define and call functions.
- **Line and Column Numbers**: The location of the function definition.
- **Parameter Types (Optional)**: A list of data types for the arguments the function expects.



Keywords

Line Number	Attribute Value
2	int
4	typedef
4	struct
5	int
6	int
11	struct
12	int
13	int
18	typedef
18	struct
19	int
20	int
25	typedef

Figure 17: Keywords Tokens Table

The Keywords Tokens Table lists keywords recognized by the programming language. Keywords have specific meanings and cannot be used as variable names. This table helps identify the core language constructs used in the code.

- **Table Headers**: "Keyword", "Line Number", (potentially) "Column Number"
- **Keywords**: Reserved words like "if", "else", "for", "while", "int", "float", etc.
- **Line and Column Numbers**: Where the keyword appears in the code.

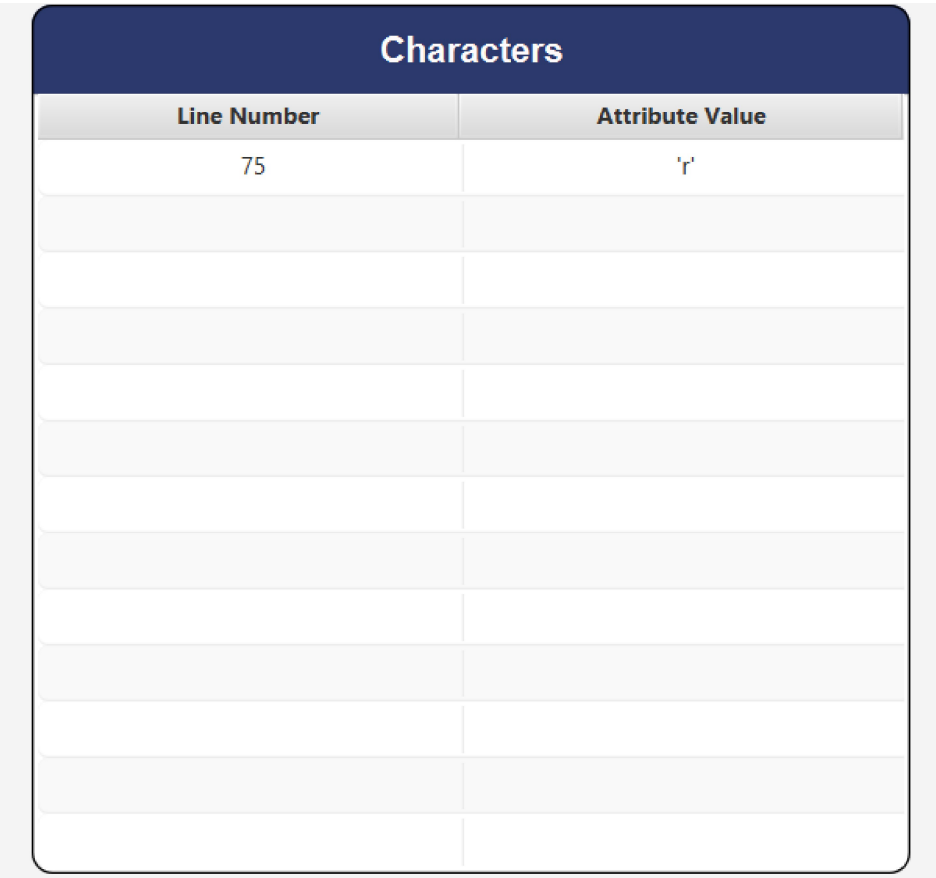
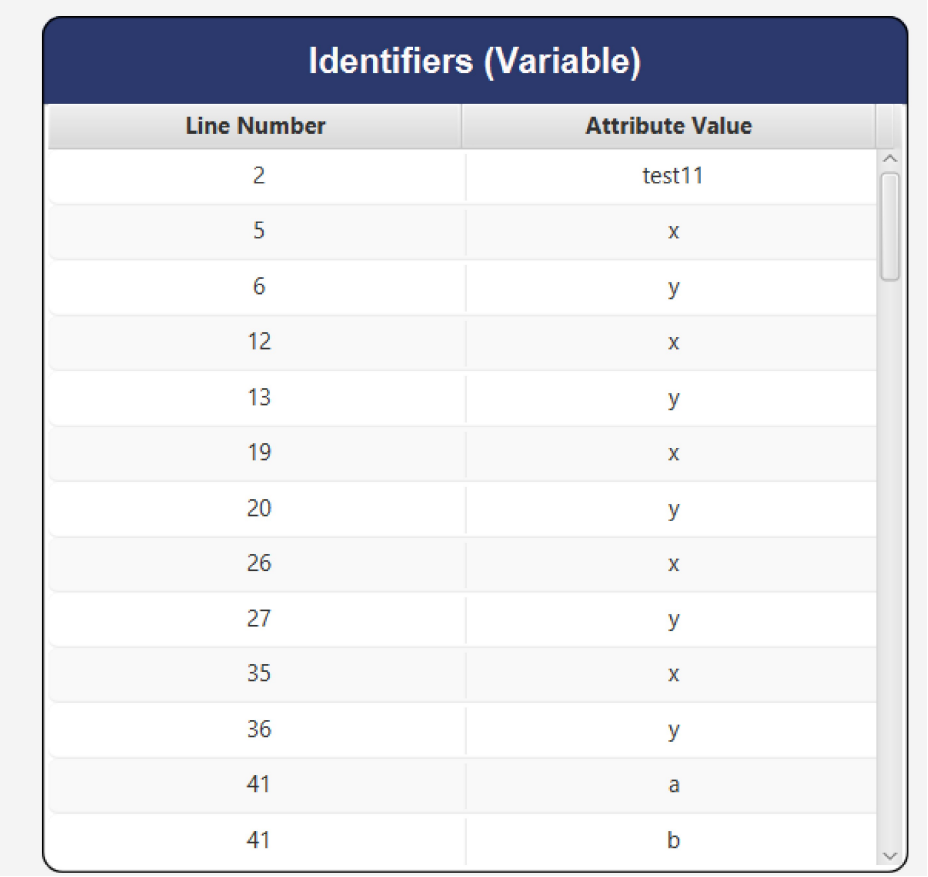


Figure 18: Characters Tokens Table

The Characters Tokens Table lists individual characters detected in the code. Characters are enclosed within single quote marks and typically represent single elements of textual data. This table aids in understanding strings and other character manipulations used within the code.

- **Table Headers**: "Character", "Line Number", (potentially) "Column Number"
- **Character**: Individual letters, numbers, symbols, or special characters found within single quotes.
- **Line and Column Numbers**: Location of the character within the source code.



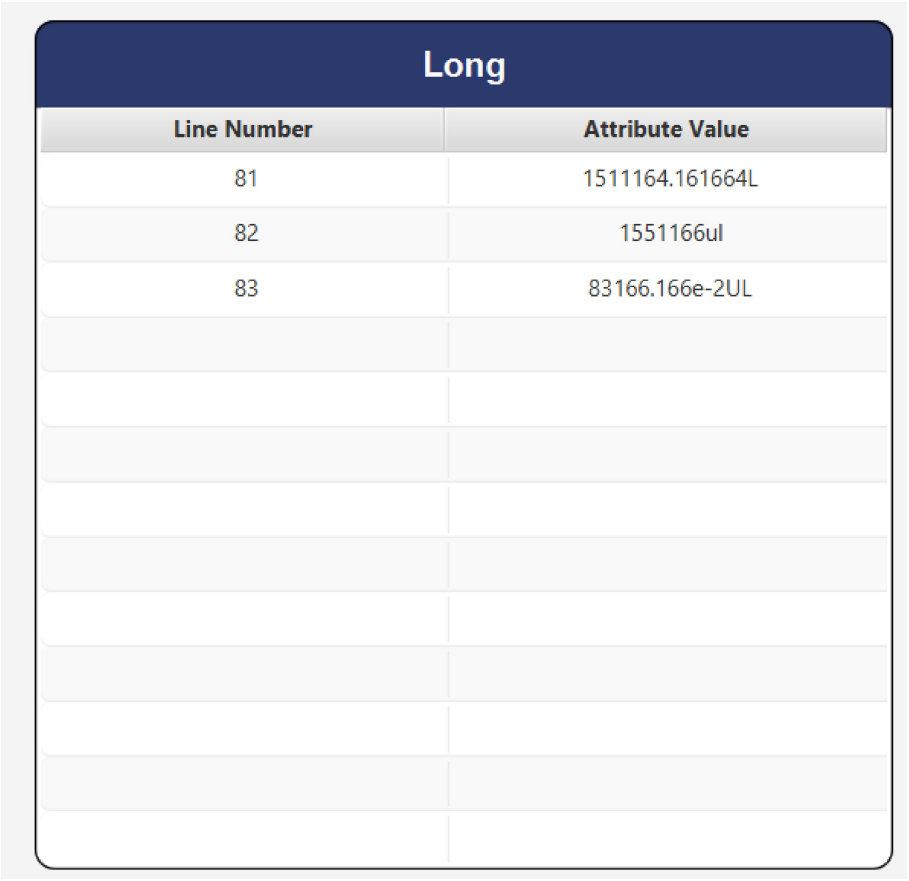
Identifiers (Variable)

Line Number	Attribute Value
2	test11
5	x
6	y
12	x
13	y
19	x
20	y
26	x
27	y
35	x
36	y
41	a
41	b

Figure 19: Identifiers (Variables) Tokens Table

The Identifiers (Variables) Tokens Table lists variable names used within the source code. Variables are used to store and manipulate data during program execution. This table likely identifies the declared variables, their data types (optional), and where they appear within the code.

- **Table Headers**: "Variable Name", "Line Number", (potentially) "Column Number", (potentially) "Data Type"
- **Variable Names**: The names used to reference data.
- **Line and Column Numbers**: The location in the code where the variable is declared or used.
- **Data Type (Optional)**: Indicates whether the variable stores integers, floating-point numbers, characters, etc.



Long

Line Number	Attribute Value
81	1511164.161664L
82	1551166ul
83	83166.166e-2UL

Figure 20: Long Tokens Table

The Long Tokens Table lists 'long' integer values found in the code. Long integers provide a larger range of whole numbers compared to regular integers, especially in older systems. This table helps identify where these larger numerical values are used within the code.

- **Table Headers**: "Long Value", "Line Number", (potentially) "Column Number"
- **Long Values**: Whole numbers with the 'long' specifier (e.g., 123456789L).
- **Line and Column Numbers**: Location of the long value in the code.

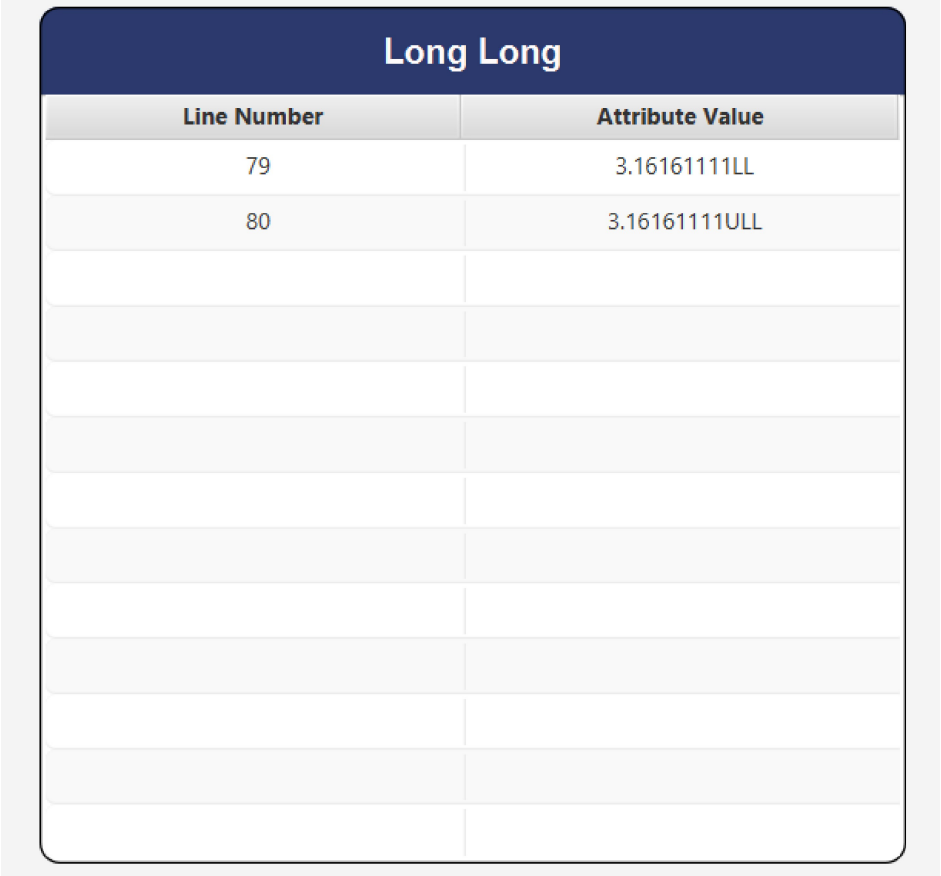
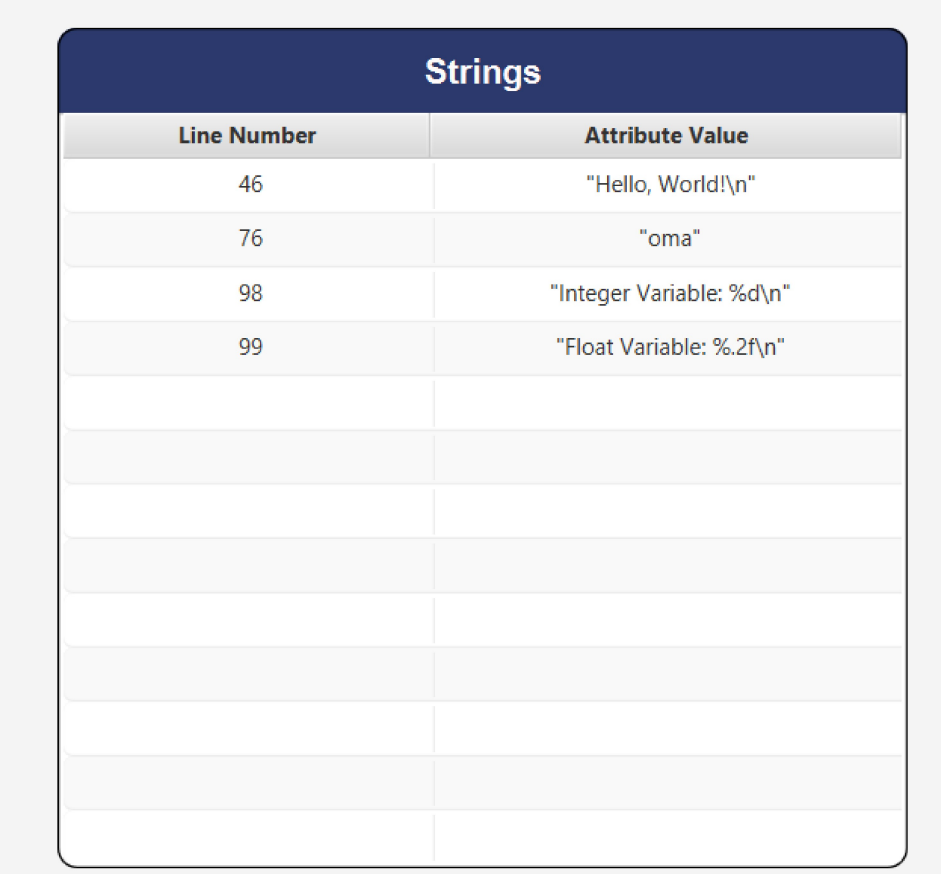


Figure 21: Long Long Tokens Table

The Long Long Tokens Table lists 'long long' integer values in the code. Long long integers offer an even wider range of whole numbers and are common in modern systems. This table helps identify where these very large numeric values are used for calculations or data storage.

- **Table Headers**: "Long Long Value", "Line Number", (potentially) "Column Number"
- **Long Long Values**: Whole numbers with the 'long long' specifier (e.g., 123456789012345LL).
- **Line and Column Numbers**: Location of the long long value within the code.



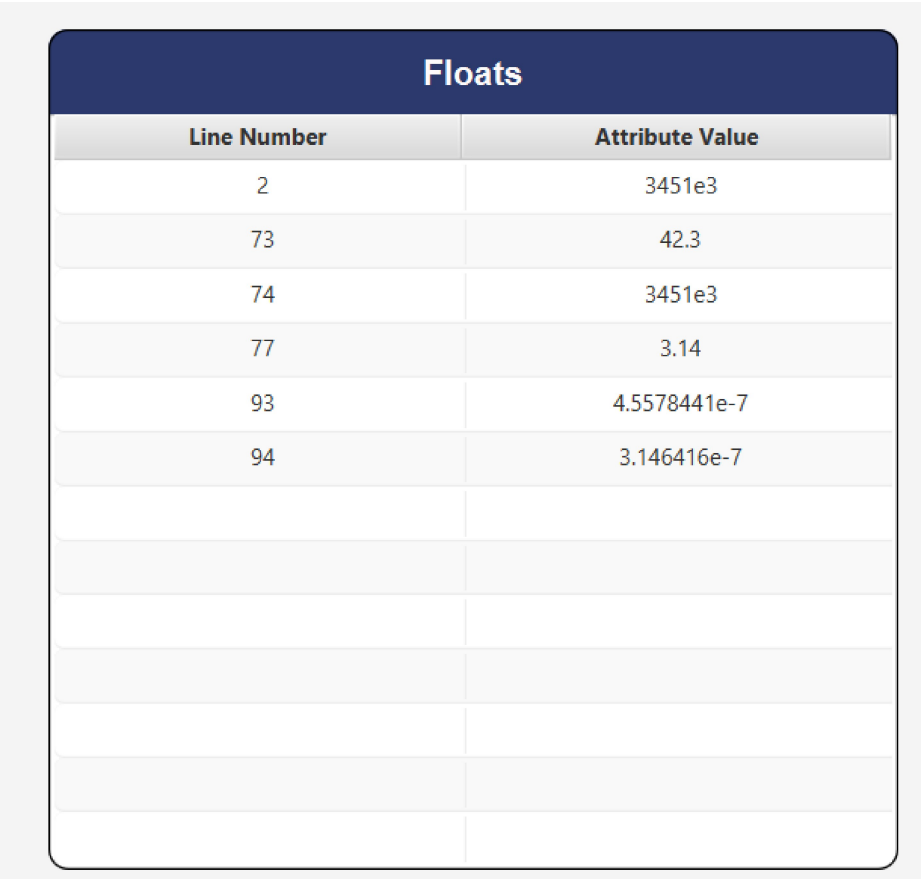
Strings

Line Number	Attribute Value
46	"Hello, World!\n"
76	"oma"
98	"Integer Variable: %d\n"
99	"Float Variable: %.2f\n"

Figure 22: Strings Tokens Table

String tokens represent textual data within the code. Understanding strings is essential for interpreting output messages, user prompts, or textual elements within the program.

- **Table Headers**: "String Value", "Line Number", (potentially) "Column Number"
- **String Values**: The actual text content enclosed within quotation marks in the source code.
- **Line and Column Numbers**: The location of the string within the source code.



Floats

Line Number	Attribute Value
2	3451e3
73	42.3
74	3451e3
77	3.14
93	4.5578441e-7
94	3.146416e-7

Figure 23: Floats Tokens Table

Floating-point tokens represent real numbers, allowing the code to handle more precise calculations compared to integer values alone.

- **Table Headers**: "Floating-Point Value", "Line Number", (potentially) "Column Number"
- **Floating-Point Values**: Decimal numbers present in the code (e.g., 3.14159, 12.5, -0.003).
- **Line and Column Numbers**: The location of the floating-point value within the source code.

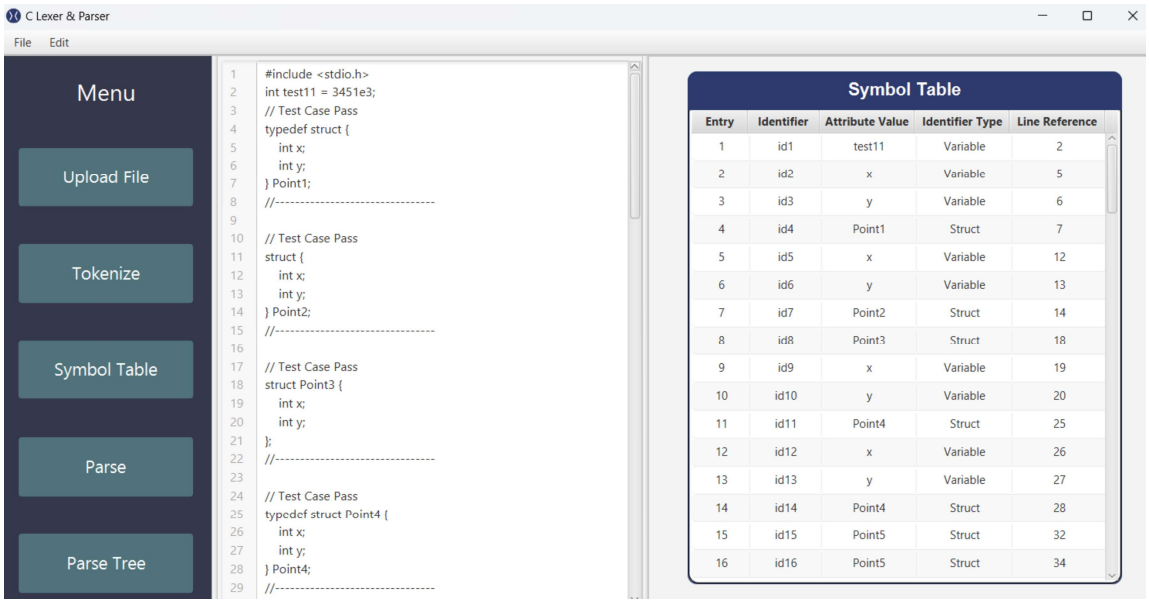
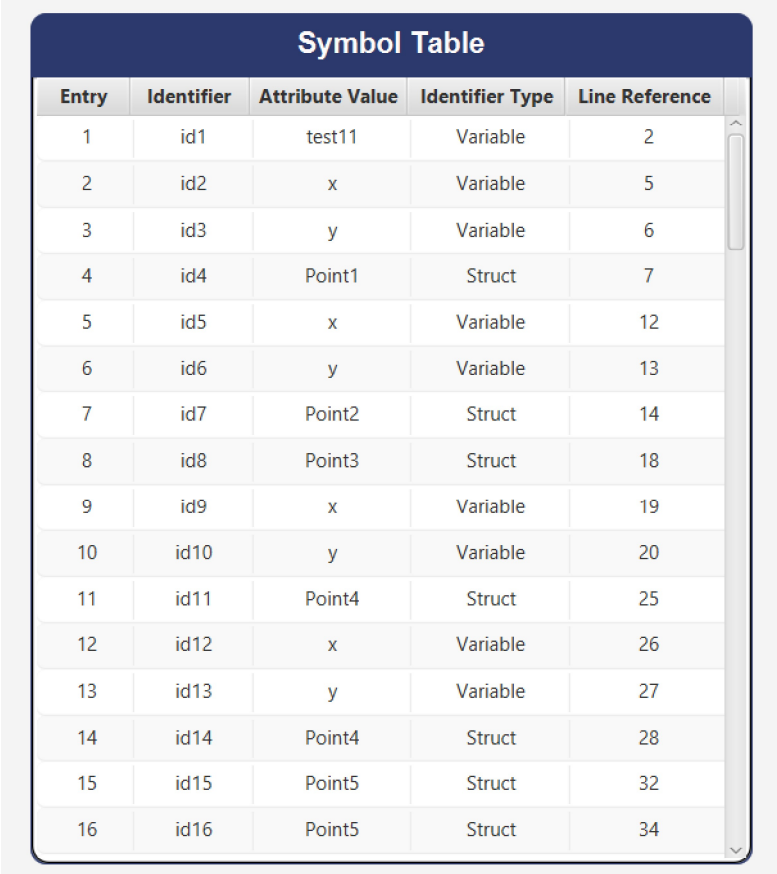


Figure 24: Symbol Table Button

The screenshot displays the symbol table window within the "C Lexer & Parser" software application. This window presents information about variables and other symbols identified during the lexical analysis stage from a C source code file.

- **Table Headers**: The top row shows the column names, which typically include "Entry", "Identifier", "Attribute Value", "Identifier Type", and "Line Reference".
- **Entry**: An identifier or sequential number assigned to each symbol within the symbol table.
- **Identifier**: The name of the variable, function, struct, or other symbol found in the code.
- **Attribute Value (Optional)**: This column might contain additional details specific to the symbol type.
- **Identifier Type**: The category or classification of the symbol, such as "Variable", "Struct", or "Function".
- **Line Reference**: The line number in the source code where the symbol was first encountered.
- **Symbol Table Entries**: The subsequent rows list the symbols identified by the lexer, along with their corresponding details.
- **Variable Information**: For variables, the "Attribute Value" column might show the variable's data type (e.g., "int", "float").
- **Struct Information**: For structs, the "Attribute Value" column might contain details about member variables within the struct.

The symbol table serves as a reference for the compiler or interpreter during subsequent stages of code processing. It provides a structured view of the symbols used within the code, aiding in memory allocation, variable lookup, and overall program analysis.



Symbol Table

Entry	Identifier	Attribute Value	Identifier Type	Line Reference
1	id1	test11	Variable	2
2	id2	x	Variable	5
3	id3	y	Variable	6
4	id4	Point1	Struct	7
5	id5	x	Variable	12
6	id6	y	Variable	13
7	id7	Point2	Struct	14
8	id8	Point3	Struct	18
9	id9	x	Variable	19
10	id10	y	Variable	20
11	id11	Point4	Struct	25
12	id12	x	Variable	26
13	id13	y	Variable	27
14	id14	Point4	Struct	28
15	id15	Point5	Struct	32
16	id16	Point5	Struct	34

Figure 25: Symbol Table

The screenshot showcases the symbol table window of the "C Lexer & Parser" application. The symbol table acts as a central repository of information about variables, functions, and other symbolic elements defined within the analyzed C source code.

- **Symbol Table Structure**: The table is organized into columns, providing specific details about each symbol:
- **Entry:** A unique identifier or index for referencing symbols.
- **Identifier:** The actual name used in the code to represent a variable, function, or other entity.
- **Attribute Value:** Additional information tied to the symbol, potentially including its data type, structure members, or other properties.
- **Identifier Type:** Classifies the symbol, indicating whether it's a variable, function, structure, etc.
- **Line Reference:** Provides the line number in the original code where the symbol was first declared or encountered.

Purpose of the Symbol Table: During compilation or interpretation, the symbol table is consulted to resolve references to variables, check for correct usage, and allocate memory. This structured representation of symbols ensures consistent interpretation of the code.