Semantic Analyzer

Project 3

Lexical Analyzer - Java

Parser - Java

Semantic Analyzer - Java

Submitted to: Professor: Llesh Miraj

April, 2024

Ashley Peleg, Ariana Contes, Andres Rodriguez, Aaron Amalraj,
Melodie Cornelly, Stephanie Sicilian

Table of Contents

Introduction

Parser

A ***parser*** is pivotal in computer science, serving as a program or part of a program that analyzes the structure of text based on a programming language. Its fundamental role is to deconstruct a sequence of symbols into component parts, strictly following the rules of a programming language. This ability renders parsers essential in a variety of contexts: from analyzing and converting source code syntax for compiler processing to executing input programs directly in interpreters without needing compilation. Additionally, parsers are instrumental in extracting structured information from unstructured text and analyzing natural language sentences to understand their structure and meaning. There are many different types of parsers based on the given situation. In our case, the parser is responsible for reading tokens from the existing Lexical Analyzer, created in order to further analyze the information by generating a parse tree (which can be generated using a Top-down or bottom-up approach if it aligns with the syntax/grammar.

Purpose/ Scope of Project

The purpose of this project is to develop a compiler for a subset of the C programming language. The compiler consists of a Lexical Analyzer that tokenizes the input C code, a Parser that builds an Abstract Syntax Tree (AST) from the tokens, and a Semantic Analyzer that performs semantic analysis on the AST. The scope of the project includes handling preprocessor directives, keywords, identifiers, constants, operators, and punctuation in the C code. Additionally, the compiler checks for variable declaration before use and performs type compatibility checks on expressions. The ultimate goal is to provide a tool that can accurately parse and analyze C code, ensuring that it adheres to the language's syntax and semantics.

The objective of this report is to thoroughly detail the design, functionality, and code organization of these two programs. This approach not only illuminates the foundational and advanced aspects of parsing and lexical analysis but also underscores their significance in programming and compiler design. sum.c introduces basic programming constructs, while LexicalAnalyzer.java offers a deeper dive into the intricate process of preparing code for compilation through tokenization.

The sections that follow will provide an in-depth look at the design and functionality of each program, shedding light on their specific code organization and their integral role in the broader context of parsing and lexical analysis.

Project Overview

Description of the C Programs:

1. sum.c:

a. Function: A user-interactive application for calculating the sum of two integers provided by the user.

b. Key Features:

1. Handles user input using the Scanner class.
2. Executes basic arithmetic operations.
3. Introduces exception handling mechanisms.
4. Demonstrates core programming constructs such as variables, data types, and control structures (loops, conditionals).

2. LexicalAnalyzer.java:

   a. Function: A more complex program designed for tokenizing C source code.

   b. Key Features:

   1. Demonstrates lexical analysis within the context of programming language compilers.
   2. Employs regular expressions to identify token types like keywords, identifiers, constants, operators, and punctuation, showcasing the importance of efficient pattern matching.
   3. Provides a practical example of source code breakdown into tokens, aiding the understanding of compilation's initial stages.

3. Compiler.java

Function: The Compiler.java tool is designed to efficiently analyze, compile, and optimize C code, transforming it into executable machine code. The tool utilizes a Lexical Analyzer to break down C code into tokens for analysis, a Parser to check the code's structure and syntax, and a Semantic Analyzer to verify the meaning of the code for correctness. In addition to the existing functionality, the tool will incorporate the following key features:

Key Features:

4. Performance Optimization: Implement various optimization techniques such as constant folding, dead code elimination, and loop invariant code motion to enhance the performance of the compiled code.

5. Translation to Machine Code: Extend the tool to generate machine-readable instructions from the parsed AST, ensuring that the compiled code can be executed by the target machine.

6. Enhanced Error Handling: Enhance the error messages to provide detailed information about syntax errors, semantic issues, and optimization warnings. This will assist developers in debugging their code more effectively.

7. Code Issue Identification: Implement functionality to identify common programming issues such as uninitialized variables, unused variables, and possible logic errors in the code. Additionally, provide suggestions for potential fixes to improve code quality.

8. Code Generation: Implement a code generation module to translate the optimized AST into executable machine code, enabling the tool to produce efficient and reliable output for the given C code.

By incorporating these additional functionalities, the Compiler.java tool aims to offer a comprehensive solution for C code analysis, compilation, and optimization, empowering developers to write high-performance and error-free programs.

Functionality

sum.c Functionality

- User Input Handling:

  The sum.c program efficiently handles user input through the utilization of the Scanner class, a standard C utility designed for parsing primitive types and strings from the input stream. Within the program, the Scanner object is instantiated to facilitate interaction with the standard input stream (System.in). This allows users to input integers directly from the console.

  Upon execution, the program prompts the user to enter the first integer, awaiting input. The Scanner class reads the input stream and parses the provided integer, storing it in the variable num1. Similarly, the program prompts the user to enter the second integer, and the Scanner class retrieves and parses this input, storing it in the variable num2.

  The interaction with the Scanner class ensures efficient and reliable user input handling, allowing for seamless integration of user-provided values into the arithmetic operations performed by the program. Additionally, the program may implement error handling mechanisms through the Scanner class to manage unexpected inputs or handle exceptions gracefully, thereby enhancing the robustness and user experience of the application.

- Arithmetic Operations:
  - Addition:
    - The program adds the two input integers using the + operator.
    - The operation is straightforward and doesn't involve complex algorithms.
    - The result is stored in the variable sum.

.

- Exception Handling:
  - Try-Catch Blocks:
    - The program implements exception handling mechanisms using try-catch blocks to manage potential errors during user input.
    - Specifically, the Scanner.nextInt() method, used to read integer input from the user, can throw InputMismatchException if the input provided is not an integer.

        - To handle this scenario, the program encloses the nextInt() method call within a try block and catches any InputMismatchException that may occur.

- Core Programming Constructs:
    - Variables:
        - Used to store input integers (num1 and num2) and the sum (sum).
        - Facilitate dynamic data storage during program execution.
    - Data Types:
        - int: Represents integer values for input and calculations.
        - Ensures consistent handling of numerical data.
    - Input Handling:
        - Relies on the Scanner class to manage user input.
        - Demonstrates how the program interacts with users.

LexicalAnalyzer.java Functionality

- Tokenization Process:

  In a lexical analyzer, "tokens" represent the smallest meaningful units of the source code, categorized into types such as keywords, identifiers, operators, punctuation symbols, literals (like numbers and strings), and other language-specific constructs. This process, known as tokenization, is vital in the initial stages of compilation, where the source code undergoes analysis and categorization into meaningful units for further processing.

  The program, LexicalAnalyzer.java, utilizes regular expressions to identify and extract tokens from the input C source code. Regular expressions serve as patterns to match character combinations in strings, enabling efficient identification of tokens based on predefined rules. The tokenization algorithm iterates through the input source code, searching for patterns that match predefined token types using regular expressions. Upon finding a match, the program identifies the corresponding token type and adds the token to the list of extracted tokens

- **Java Packages Utilized:**

1. **java.util.regex.Matcher, and java.util.regex.Pattern** are used to work with regular expressions. Patterns are used for creating the regex pattern we are looking for, and it has a method called compile used for indicating the pattern. A matcher is a package that has a method called find to see if the given input matches the pattern, and other methods.

2. **java.io.IOException** is used for exception handling, in our program(LexicalAnalyzer.java) to catch an error and print out that error, using the getMessage() method, and throw an IllegalStateException in the case that we get an unrecognizable token.

3. **java.nio.file.Files and java.nio.file.Paths** are used to read the file sum.c in its respective path.

4. **java.util.ArrayList** is used to store all of the tokens found.

class Token defines a token, by having a TokenType (which is from the enum TokenType), and its associated value which is a String.

public class LexicalAnalyzer: *which contains*:

1. **private static final** (private-restricting visibility only within the class, static meaning these variables are for the whole class and not just an instance and final meaning that once initialized these values cannot be changed) defines a string array called **patterns** which define the **patterns to be recognized using regex** (once string/element pattern in array) for each tokentype.
2. **public static ArrayList<Token> tokenize(String program)** is a static method called tokenize, that takes in a String called ***"program"*** and returns an ArrayList of Token objects, which were defined in the class Token. It does this firstly, by creating an ArrayList of Token objects called tokens. Then, Pattern pattern = Pattern.compile(String.join("|", patterns)); creates a Pattern object by using the compile method to join the patterns string array using **| as a delimiter** for each element in that array. | is used as a delimiter because in Java regex it means to find "a match for any one of the patterns separated by | as in: cat|dog|fish". After this a matcher object is created by using our pattern string which is the one we just created and using the matcher object to create a matcher object by comparing pattern to String program which is the input that is received in this function. A while loop is then used to see if the matcher object which has information on the String ***program*** has any of the tokens from ***pattern*** the while condition uses the find() method. If find () is true it goes into the while loop it uses a method called group which returns the substring from ***program*** that matched with ***pattern***. It then checks this substring to find its corresponding tokentype, if it does not apply to any of the tokentypes regex or a matching error occurs an **IllegalStateException** is thrown. The substring if it has a tokentype is added to the arraylist and the process repeats iterating over each match with the program string by using the matcher object.
3. **Main method:** defines a file path. Uses a try catch statement first, Files.readAllBytes(Paths.get(filePath)) reads the path object as a byte array which is then converted to a string object (called program) because it is initialized as that. Then, an arraylist of tokens is created and assigned to a call to the method tokenize with our string program (which is the information from our file). Then a for loop is used to print each token with its type and value. If this does not work for whatever reason, there is a catch statement that catches exceptions/errors.

- **Token Types: enum TokenType is used to define the different token types.**
  - KeyWords: Keywords are reserved words in Java with predefined meanings that cannot be used as identifiers.They are identified via a regular expression pattern that matches specific keywords. Words Identified: int, float, char, if, else, while, for, and return

  - Identifiers: Identifiers are names assigned to variables, methods, classes, and other entities in Java programs. They are recognized as sequences of characters that must begin with a letter or underscore, followed by zero or more letters, digits, or underscores. In the LexicalAnalyzer.java program, identifiers are identified using the following regular expression pattern:

  - \\b[a-zA-Z_]\\w*\\b
  - Tokens conforming to this pattern, representing identifiers, are categorized as identifier tokens during the tokenization process.
  - Elements above explained:
    - \\b: Represents a word boundary, ensuring that the identifier starts and ends with a complete word.
    - [a-zA-Z_]: Matches a single character that is either a lowercase letter (a-z), an uppercase letter (A-Z), or an underscore (_). This ensures that the identifier starts with a letter or underscore.
    - \\w*: Matches zero or more word characters, including letters, digits, and underscores. This allows identifiers to contain letters, digits, or underscores after the first character.

- Constants: Constants represent fixed values in Java programs, such as numeric literals.
  - Identified as sequences of digits (\d+).
  - Tokens conforming to this pattern, representing constants, are categorized as constant tokens during the tokenization process.

- Operators: Operators are symbols used to perform operations on operands in Java programs. Tokens conforming to patterns representing these symbols are categorized as operator tokens during the tokenization process. Symbols include:
  - Arithmetic operators: +, -, *, /
  - Assignment operator: =
  - Comparison operators: ==, !=, <, >, <=, >=
  - Logical operators: &&, ||

- Punctuation: Punctuation symbols are special characters used to punctuate and structure Java code. Tokens conforming to patterns representing these symbols are categorized as punctuation tokens during the tokenization process. Symbols include:
  - Semicolons: ;
  - Commas: ,
  - Parentheses: (,)
  - Braces: {, }
  - Square brackets: [,]

Compiler.java Functionality

The functions and key features of compiler.java include the following:

1. Lexical analysis: This function involves breaking down the input Java code into a sequence of tokens, such as identifiers, keywords, operators, and literals.

2. Syntax analysis: This function checks the sequence of tokens for correct syntax based on the rules of the C language, ensuring that the code is structured correctly.

3. Semantic analysis: This function involves checking the meaning of the code to ensure that it adheres to the semantic rules of the C language, such as type compatibility and variable declarations.

4. Optimization: Compiler.java includes optimization techniques to improve the performance of the generated machine code, such as constant folding, dead code elimination, and loop optimizations.

5. Code generation: This function involves translating the parsed and optimized code into executable machine code that can be run on a specific hardware platform.

6. Error handling: The compiler may include robust error-handling mechanisms to provide informative error messages to developers in case of syntax or semantic errors in their code.

7. Debugging support: Compiler.java may also offer debugging features, such as line number mappings and debugging symbols, to help developers identify and fix issues in their code.

For Semantic Analysis context free grammar :

8. Tokenization: Compiler.java performs tokenization on the input C code, breaking it down into smaller parts (tokens) for further analysis and processing.

9. Structure Verification: The tool checks if the C code adheres to the correct structure and syntax rules specified by the C language, ensuring that it is well-formed and syntactically correct.

10. Performance Analysis: Compiler.java analyzes the code for performance bottlenecks and inefficiencies, providing optimizations to enhance the overall execution speed and efficiency of the compiled code.

11. Machine Code Translation: The tool translates the optimized C code into machine-readable instructions that can be executed directly by the target hardware platform, ensuring compatibility and efficiency.

12. Issue Resolution: Compiler.java assists in identifying and resolving issues in the code, providing suggestions and guidance to improve the quality and correctness of the C code being compiled.

13. Enhanced Code Transformation: The tool may implement advanced code transformation techniques such as loop unrolling, function inlining, and register allocation to further optimize the performance of the compiled code.

The semantic analyzer in Compiler.java plays a crucial role in ensuring that the input C code adheres to the language's semantic rules and guidelines. By analyzing the meaning and context of the code, this component checks for issues such as type compatibility, variable declarations, and overall code coherence. It helps prevent logical errors, identify potential issues related to variable usage, and ensure that the code functions correctly in different contexts. The semantic analyzer acts as a key tool in validating the correctness and integrity of the C code before proceeding to the optimization and code generation stages, ultimately contributing to the production of efficient and error-free compiled code.

Java Packages Utilized:

1. import java.util.ArrayList;

This package allows you to use the ArrayList data structure in your Java code.

2. import java.util.List;

This package works with List implementations such as ArrayList, LinkedList, etc. This package allows you to use List-specific methods and functionalities in your code.

3. import java.util.regex.Matcher;

This package provides classes and interfaces for dealing with regular expressions in Java.

4. import java.util.regex.Pattern;

The Pattern class is another class from the java.util.regex package that represents a compiled version of a regular expression pattern.

5. import java.nio.file.Files;

This package provides classes and interfaces for file I/O operations in Java.

6. import java.nio.file.Paths;

The Paths class is part of the java.nio.file package and provides methods for converting a string or URI to a Path object.

7. import java.io.IOException;

This package provides classes for handling input/output operations in Java.

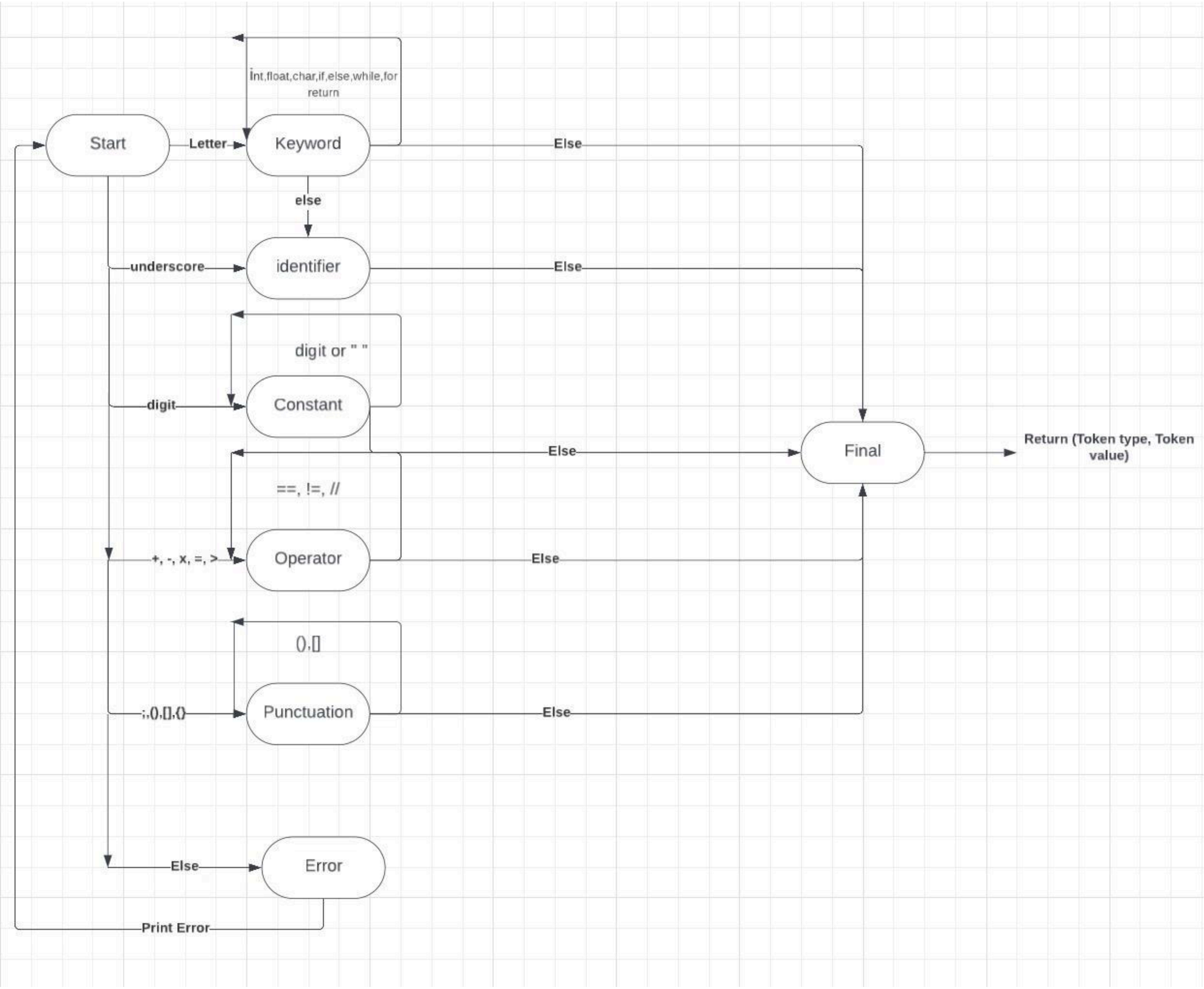
8. Import java.util.HashSet;

This line imports the HashSet class from the java.util package, allowing the use of HashSet objects in the Java program.

9. Import java.util.Set;

This line imports the Set interface from the java.util package, allowing the use of Set interface and its implementations in the Java program.

State Diagram



Conclusion

The development of a parser and compiler for C code, written in Java is crucial in ensuring the efficient analysis, compilation, and optimization of Java programs. The Compiler.java program breaks down C code into tokens, checks for syntactical correctness, and constructs an Abstract Syntax Tree (AST) to represent the hierarchical structure of the code. Through semantic analysis, optimization, and code generation, the compiler ensures that the C code generates high-performance machine-readable instructions. The use of Java packages such as ArrayList, List, regex Matcher and Pattern, and NIO files provides functionalities for file I/O operations, regular expression handling, and data structure manipulation within the compiler. The development of a compiler like Compiler.java improves the development process, enhances code performance, and provides valuable error-handling and debugging support for C developers. The project's focus is creating a user-friendly interface and optimizing the compilation process, ultimately aiming to improve the reliability and efficiency of C applications.

Overall, this project underscored the importance of lexical analysis in the parsing process. The tokenization example in LexicalAnalyzer.java provided a useful reference for the techniques involved in this initial stage of compiling source code. The report detailed the key components of the program - the matching rules for different token types, the categorization process, and the role played by regular expressions in efficient pattern matching.

In addition to the parsing process described in the conclusion, a semantic analyzer is another crucial component in the compilation process that was not extensively discussed in this context. The semantic analyzer, which is typically the phase following lexical and syntax analysis, ensures that the code adheres to the defined rules of the programming language, making sure that identifiers are used correctly, types are compatible, and that the code follows the semantic rules of the language.

In the context of the Compiler.java program developed in this project, the semantic analysis phase would involve further checking the semantic correctness of the C code after it has been parsed and structured into an Abstract Syntax Tree. This phase may involve type checking, scope analysis, symbol table management, and other tasks necessary to verify the meaning and intent of the code.

The semantic analyzer plays a crucial role in ensuring that the compiled code is not only syntactically correct but also semantically valid and compliant with the language specifications. By incorporating semantic analysis into the compiler development process, the project can guarantee the production of optimized and error-free machine-readable instructions for C programs.

Through the development and analysis of these programs, the project served as an effective learning tool to understand core programming concepts, as well as gain insight into advanced parsing mechanisms like lexical analysis and recursive descent parsers. It illustrated the practical application of foundational and complex aspects of parsers within the domain of C programming.

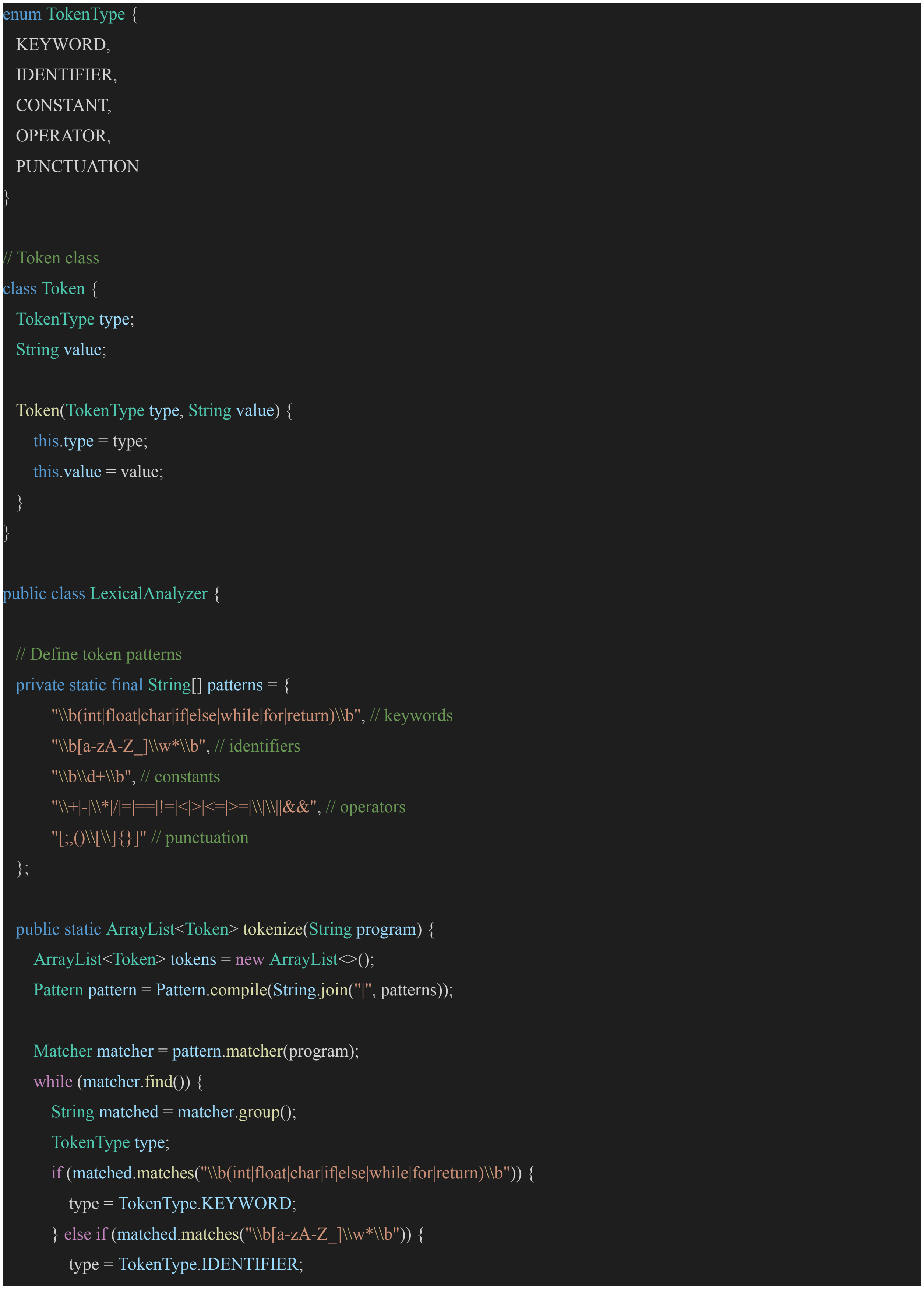
Appendix

1. Source Code for sum.c:

```c
#include <stdio.h>

int main() {
  int num1, num2, sum;

  printf("Enter the first integer: ");
  scanf("%d", &num1);

  printf("Enter the second integer: ");
  scanf("%d", &num2);

  sum = num1 + num2;

  printf("The sum of %d and %d is %d\n", num1, num2, sum);

  return 0;
}
```

2. Source Code for Lexical Analyzer.java: (Project #1)

```java
import java.util.regex.Matcher;
import java.util.regex.Pattern;
import java.nio.file.Files;
import java.nio.file.Paths;
import java.io.IOException;
import java.util.ArrayList;



// Token types
```



```
enum TokenType {
    KEYWORD,
    IDENTIFIER,
    CONSTANT,
    OPERATOR,
    PUNCTUATION
}

// Token class
class Token {
    TokenType type;
    String value;

    Token(TokenType type, String value) {
        this.type = type;
        this.value = value;
    }
}

public class LexicalAnalyzer {

    // Define token patterns
    private static final String[] patterns = {
            "\\b(int|float|char|if|else|while|for|return)\\b", // keywords
            "\\b[a-zA-Z_]\\w*\\b", // identifiers
            "\\b\\d+\\b", // constants
            "\\+|-|\\*|/|=|==|!=|<|>|<=|>=|\\|\\||&&", // operators
            "[;,()\\[\\]{}]" // punctuation
    };

    public static ArrayList<Token> tokenize(String program) {
        ArrayList<Token> tokens = new ArrayList<>();
        Pattern pattern = Pattern.compile(String.join("|", patterns));

        Matcher matcher = pattern.matcher(program);
        while (matcher.find()) {
            String matched = matcher.group();
            TokenType type;
            if (matched.matches("\\b(int|float|char|if|else|while|for|return)\\b")) {
                type = TokenType.KEYWORD;
            } else if (matched.matches("\\b[a-zA-Z_]\\w*\\b")) {
                type = TokenType.IDENTIFIER;
            } else if (matched.matches("\\b\\d+\\b")) {
                type = TokenType.CONSTANT;
            } else if (matched.matches("\\+|-|\\*|/|=|==|!=|<|>|<=|>=|\\|\\||&&")) {
                type = TokenType.OPERATOR;
            } else if (matched.matches("[;,()\\[\\]{}]")) {
                type = TokenType.PUNCTUATION;
            } else {
                throw new IllegalStateException("Unrecognized token: " + matched);
            }
            tokens.add(new Token(type, matched));
        }

        return tokens;
    }

    public static void main(String[] args) {
        String filePath = "/Users/arianacontes/CS361/sum.c";

        try {
            String program = new String(Files.readAllBytes(Paths.get(filePath)));

            ArrayList<Token> tokens = tokenize(program);

            for (Token token : tokens) {
                System.out.println("(" + token.type + ", " + token.value + ")");
            }
        } catch (IOException e) {
            System.err.println("Error reading file: " + e.getMessage());
        }
    }
}
```

Sample Output from Lexical Analyzer.java:

```
(IDENTIFIER, include)
(OPERATOR, <)
(IDENTIFIER, stdio)
(IDENTIFIER, h)
(OPERATOR, >)
(KEYWORD, int)
(IDENTIFIER, main)
```

```
(PUNCTUATION, ()
(PUNCTUATION, ))
(PUNCTUATION, {)
(KEYWORD, int)
(IDENTIFIER, num1)
(PUNCTUATION, ,)
(IDENTIFIER, num2)
(PUNCTUATION, ,)
(IDENTIFIER, sum)
(PUNCTUATION, ;)
(IDENTIFIER, printf)
(PUNCTUATION, ()
(IDENTIFIER, Enter)
(IDENTIFIER, the)
(IDENTIFIER, first)
(IDENTIFIER, integer)
(PUNCTUATION, ))
(PUNCTUATION, ;)
(IDENTIFIER, scanf)
(PUNCTUATION, ()
(IDENTIFIER, d)
(PUNCTUATION, ,)
(IDENTIFIER, num1)
(PUNCTUATION, ))
(PUNCTUATION, ;)
(IDENTIFIER, printf)
(PUNCTUATION, ()
(IDENTIFIER, Enter)
(IDENTIFIER, the)
(IDENTIFIER, second)
(IDENTIFIER, integer)
(PUNCTUATION, ))
(PUNCTUATION, ;)
(IDENTIFIER, scanf)
(PUNCTUATION, ()
(IDENTIFIER, d)
(PUNCTUATION, ,)
(IDENTIFIER, num2)
(PUNCTUATION, ))
(PUNCTUATION, ;)
(IDENTIFIER, sum)
(OPERATOR, =)
(IDENTIFIER, num1)
(OPERATOR, +)
(IDENTIFIER, num2)
(PUNCTUATION, ;)
(IDENTIFIER, printf)
(PUNCTUATION, ()
(IDENTIFIER, The)
(IDENTIFIER, sum)
(IDENTIFIER, of)
(IDENTIFIER, d)
(IDENTIFIER, and)
(IDENTIFIER, d)
(IDENTIFIER, is)
(IDENTIFIER, d)
(IDENTIFIER, n)
(PUNCTUATION, ,)
(IDENTIFIER, num1)
(PUNCTUATION, ,)
(IDENTIFIER, num2)
(PUNCTUATION, ,)
(IDENTIFIER, sum)
(PUNCTUATION, ))
(PUNCTUATION, ;)
(KEYWORD, return)
(CONSTANT, 0)
(PUNCTUATION, ;)
(PUNCTUATION, })
```

Updated Sample code for Compile.java

```
//to compile do:javac Compiler.java to run do:java Compiler sum.c

//Compiler.java updated code for semantic analyzer:

import java.util.ArrayList;

import java.util.List;
import java.util.Set;
import java.util.HashSet;
import java.util.regex.Matcher;
import java.util.regex.Pattern;
import java.nio.file.Files;
import java.nio.file.Paths;
import java.io.IOException;

// Token types
enum TokenType {
PREPROCESSOR_DIRECTIVE,
KEYWORD,
IDENTIFIER,
CONSTANT,
OPERATOR,
PUNCTUATION
}

// Token class
class Token {
TokenType type;
String value;

Token(TokenType type, String value) {
this.type = type;
this.value = value;
```

```
}
}

// Lexical Analyzer class
class LexicalAnalyzer {
public static ArrayList<Token> tokenize(String code) {
ArrayList<Token> tokens = new ArrayList<>();
// Define token patterns
String[] patterns = {
"#include <\\w+\\.h>",
"\\b(int|float|char|if|else|while|for|return)\\b", // keywords
"\\b[a-zA-Z_]\\w*\\b", // identifiers
"\\b\\d+\\b", // constants
"\\+|-|\\*|/|=|==|!=|<|>|<=|>=|\\|\\||&&", // operators
"[;,()\\[\\]{}]" // punctuation
};
Pattern pattern = Pattern.compile(String.join("|", patterns));

Matcher matcher = pattern.matcher(code);
while (matcher.find()) {
String matched = matcher.group();
TokenType type;
if (matched.matches("#include <\\w+\\.h>")) {
type = TokenType.PREPROCESSOR_DIRECTIVE;
} else if (matched.matches("\\b(int|float|char|if|else|while|for|return)\\b")) {
type = TokenType.KEYWORD;
} else if (matched.matches("\\b[a-zA-Z_]\\w*\\b")) {
type = TokenType.IDENTIFIER;
} else if (matched.matches("\\b\\d+\\b")) {
type = TokenType.CONSTANT;
} else if (matched.matches("\\+|-|\\*|/|=|==|!=|<|>|<=|>=|\\|\\||&&")) {
type = TokenType.OPERATOR;
} else if (matched.matches("[;,()\\[\\]{}]")) {
type = TokenType.PUNCTUATION;
} else {
```

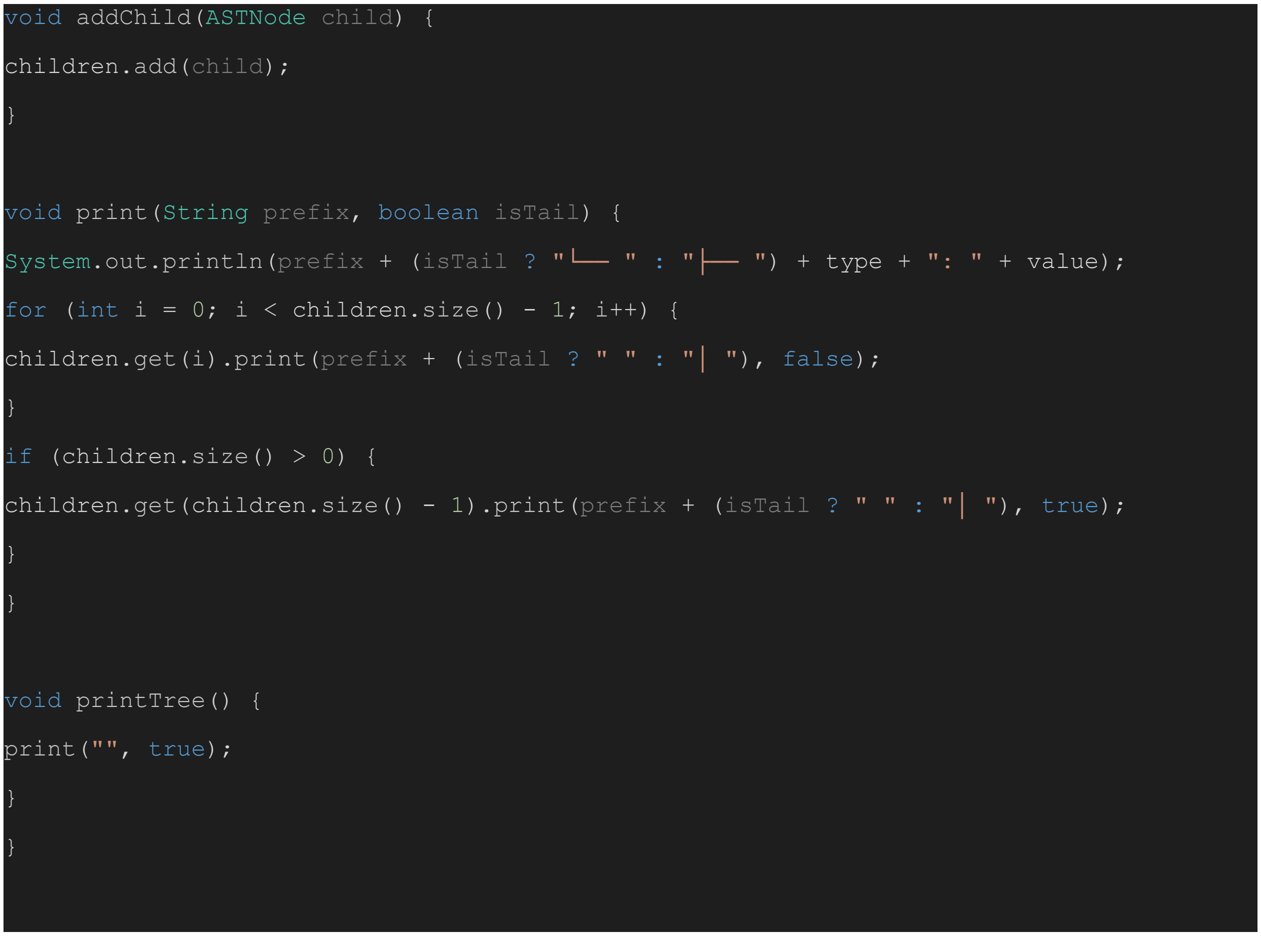
```
throw new IllegalStateException("Unrecognized token: " + matched);
}
tokens.add(new Token(type, matched));
}

return tokens;
}
}

// Parser class
class Parser {
private ASTNode parseProgram(ArrayList<Token> tokens) {
ASTNode root = new ASTNode("Program", "");

ASTNode currentBlock = root; // Track the current block being filled

// Iterate through tokens to build the AST
for (int i = 0; i < tokens.size(); i++) {
Token token = tokens.get(i);
if (token.type == TokenType.PREPROCESSOR_DIRECTIVE) {
root.addChild(new ASTNode("Preprocessor Directive", token.value));
} else if (token.value.equals("int") && i + 1 < tokens.size() && tokens.get(i +
1).value.equals("main")) {
// Found the main function declaration, create a block for its body
ASTNode mainBlock = new ASTNode("Block", "");
currentBlock.addChild(mainBlock);
currentBlock = mainBlock;
} else if (token.value.equals("{")) {
// Found the start of a new block, create a nested block
ASTNode newBlock = new ASTNode("Block", "");
currentBlock.addChild(newBlock);
currentBlock = newBlock;
} else if (token.value.equals("}")) {
// Found the end of the current block, move back to the parent block
if (currentBlock != root) {
currentBlock = getParentBlock(root, currentBlock);
}
} else {
// Add the token as a child of the current block
currentBlock.addChild(new ASTNode(token.type.toString(), token.value));
}
}

return root;
}

private ASTNode getParentBlock(ASTNode root, ASTNode currentBlock) {
// Traverse the tree to find the parent block of the current block
if (root.children.contains(currentBlock)) {
return root;
}
for (ASTNode child : root.children) {
ASTNode parent = getParentBlock(child, currentBlock);
if (parent != null) {
return parent;
}
}
return null;
}

public ASTNode buildAST(ArrayList<Token> tokens) {
return parseProgram(tokens);
}

// Semantic analysis methods
public boolean performSemanticAnalysis(ASTNode root) {
// Perform semantic analysis here
// Example: Check for variable declaration before use
return checkVariableDeclarationBeforeUse(root) && checkTypeCompatibility(root);
}
```

```
// Method to check if variables are declared before use
private boolean checkVariableDeclarationBeforeUse(ASTNode node) {
// Track declared variables
Set<String> declaredVariables = new HashSet<>();

// Track semantic analysis result
boolean success = true;

// Traverse the AST
for (ASTNode child : node.children) {
if (child.type.equals("Declaration")) {
// If a variable is declared, add it to declaredVariables set
declaredVariables.add(child.children.get(0).value);
} else if (child.type.equals("Identifier")) {
// If an identifier is found, check if it's declared
if (!declaredVariables.contains(child.value)) {
System.err.println("Error: Variable '" + child.value + "' used before declaration.");
success = false;
}
}

// Recursively check children
success &= checkVariableDeclarationBeforeUse(child);
}

return success;
}

// Method to perform type compatibility checks
private boolean checkTypeCompatibility(ASTNode node) {
// Track semantic analysis result
boolean success = true;

// Traverse the AST
for (ASTNode child : node.children) {
```

```
// If the node is an expression, perform type checking
if (child.type.equals("Expression")) {
// Check if both operands of binary operators are of compatible types
if (!checkExpressionTypeCompatibility(child)) {
success = false;
}
}

// Recursively check children
success &= checkTypeCompatibility(child);
}

return success;
}

// Method to check type compatibility in expressions
private boolean checkExpressionTypeCompatibility(ASTNode node) {
// Track semantic analysis result
boolean success = true;

// For simplicity, assume binary expressions consist of two children
if (node.children.size() != 3) { // Operator and two operands
System.err.println("Error: Invalid expression structure.");
return false;
}

// Extract operator and operand nodes
ASTNode operatorNode = node.children.get(1);
ASTNode leftOperand = node.children.get(0);
ASTNode rightOperand = node.children.get(2);

// Check if operator and operands are of compatible types
String operator = operatorNode.value;
if (operator.equals("+") || operator.equals("-") || operator.equals("*") ||
operator.equals("/")) {
// Arithmetic operations: operands must be numeric
```

```
if (!isNumericType(leftOperand) || !isNumericType(rightOperand)) {
System.err.println("Error: Type mismatch in arithmetic expression.");
success = false;
}
}

return success;
}

// Helper method to check if a node represents a numeric type
private boolean isNumericType(ASTNode node) {
// For simplicity, assume numeric types are 'int' and 'float'
return node.value.equals("int") || node.value.equals("float");
}
}

// Compiler class
public class Compiler {
public static void main(String[] args) {
if (args.length != 1) {
System.err.println("Usage: java Compiler <path_to_c_file>");
return;
}

String filePath = args[0];
try {
// Read the content of the C file
List<String> lines = Files.readAllLines(Paths.get(filePath));
StringBuilder codeBuilder = new StringBuilder();
for (String line : lines) {
codeBuilder.append(line).append("\n");
}
String code = codeBuilder.toString();

// Tokenize the C code
```

```
ArrayList<Token> tokens = LexicalAnalyzer.tokenize(code);

// Build the Abstract Syntax Tree (AST)
Parser parser = new Parser();
ASTNode ast = parser.buildAST(tokens);

// Perform semantic analysis
boolean semanticAnalysisSuccess = parser.performSemanticAnalysis(ast);

if (semanticAnalysisSuccess) {
// Print AST
System.out.println("Abstract Syntax Tree (AST):");
ast.printTree();
System.out.println("Semantic analysis successful. Parsing successful");
} else {
System.err.println("Semantic analysis failed. AST will not be printed.");
}
} catch (IOException e) {
System.err.println("Error reading file: " + e.getMessage());
}
}
}

// AST node classes
class ASTNode {
String type;
String value;
ArrayList<ASTNode> children;

ASTNode(String type, String value) {
this.type = type;
this.value = value;
this.children = new ArrayList<>();
}
```



```java
void addChild(ASTNode child) {
children.add(child);
}

void print(String prefix, boolean isTail) {
System.out.println(prefix + (isTail ? "└── " : "├── ") + type + ": " + value);
for (int i = 0; i < children.size() - 1; i++) {
children.get(i).print(prefix + (isTail ? "    " : "│   "), false);
}
if (children.size() > 0) {
children.get(children.size() - 1).print(prefix + (isTail ? "    " : "│   "), true);
}
}

void printTree() {
print("", true);
}
}
```

Sample Output of Compiler.java

```
PREPROCESSOR_DIRECTIVE: #include <stdio.h>
KEYWORD: int
IDENTIFIER: main
PUNCTUATION: (
PUNCTUATION: )
PUNCTUATION: {
KEYWORD: int
IDENTIFIER: a
OPERATOR: =
CONSTANT: 5
PUNCTUATION: ;
KEYWORD: int
IDENTIFIER: b
OPERATOR: =
CONSTANT: 10
PUNCTUATION: ;
KEYWORD: int
IDENTIFIER: sum
OPERATOR: =
IDENTIFIER: a
OPERATOR: +
IDENTIFIER: b
PUNCTUATION: ;
IDENTIFIER: printf
PUNCTUATION: (
IDENTIFIER: Sum
IDENTIFIER: d
PUNCTUATION: ,
IDENTIFIER: sum
PUNCTUATION: )
PUNCTUATION: ;
KEYWORD: return
CONSTANT: 0
PUNCTUATION: ;
PUNCTUATION: }
Semantic analysis successful. Parsing successful
```

Sources:

Intro to Parsers