DESIGNING A SCANNER FOR C LANGUAGE

Abdul Ghani Khan - 2020005
Muhammad Bilal Rafique - 2020103

Table of Contents

Introduction

Compiler

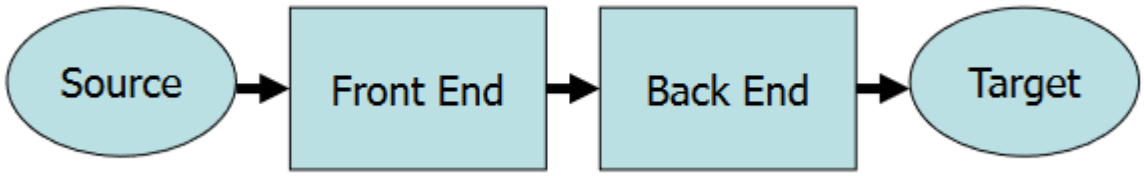
A compiler is a program that can read a program in one language - the source language - and translate it to an equivalent program in another language - the target language. An important role of the compiler is to detect any errors in the source program during the translation process.

Structure of a compiler

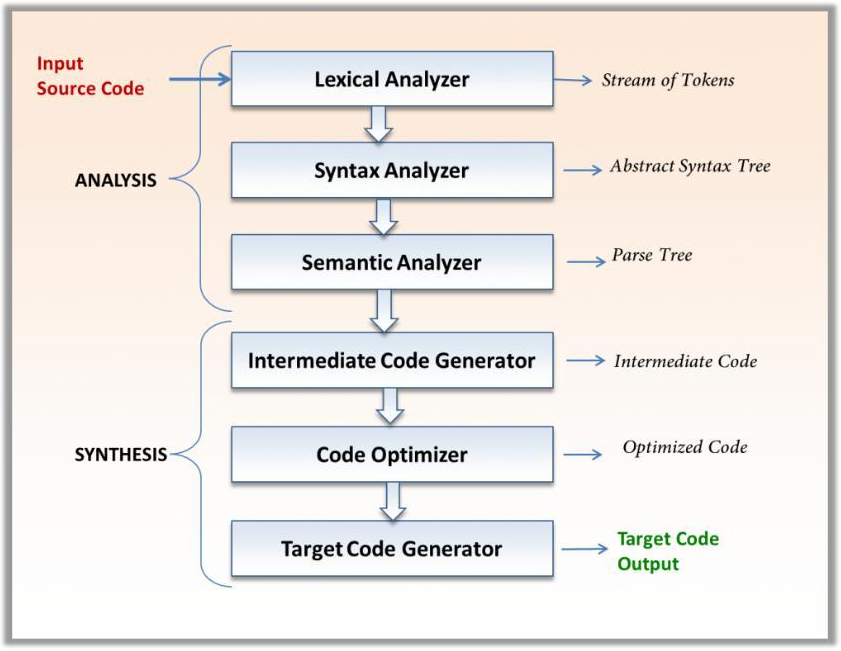
There are two parts involved in the translation of a program in the source language into a semantically equivalent target program: analysis and synthesis.

The analysis part breaks up the source program into constituent pieces and imposes a grammatical structure on them. It then uses this structure to create an intermediate representation of the source program. The analysis part also collects information about the source program and stores it in a data structure called a symbol table, which is passed along with the intermediate representation to the synthesis part.

The synthesis part constructs the desired target program from the intermediate representation and the information in the symbol table. The analysis part is often called the front end of the compiler and the synthesis part is called the back end.



Phases of compilation



The compilation process operates as a sequence of phases each of which transforms one representation of the source program to another.

- The first phase of a compiler is called **lexical analysis or scanning.**The lexical analyzer reads the stream of characters making up the source program and groups the characters into meaningful sequences called lexemes For each lexeme, the lexical analyzer produces as output a token that it passes on to the subsequent phase, syntax analysis.
- The second phase of the compiler is **syntax analysis or parsing.**The parser uses the tokens produced by the lexical analyzer to create a tree-like intermediate representation that depicts the grammatical structure of the token stream.
- The third phase is the **semantic analysis**. The semantic analyzer uses the syntax tree and the information in the symbol table to check the source program for semantic consistency with the language definition. An important part of semantic analysis is type checking, where the compiler checks that each operator has matching operands.
- After syntax and semantic analysis of the source program, many compilers generate an explicit low-level or machine-like **intermediate code representation**.
- The machine-independent **code-optimization**phase attempts to improve the intermediate code so that better target code will result.
- The last phase is the **code generation**. The code generator takes as input an intermediate representation of the source program and maps it into the target language

Lexical Analysis

As the first phase of a compiler, the main task of the lexical analyzer is to read the input characters of the source program, group them into lexemes, and produce as output a sequence of tokens for each lexeme in the source program.

The lexical analyzer maintains a data structure called as the symbol table. When the lexical analyzer discovers a lexeme constituting an identifier, it enters that lexeme into the symbol table.

The lexical analyzer performs certain other tasks besides identification of lexemes. One such task is stripping out comments and whitespace. Another task is correlating error messages generated by the compiler with the source program

Designing a scanner for C language

We built a lexer to perform lexical analysis on a subset of the C programming language. Lexer is a lexical analyzer generator that takes in a set of descriptions of possible tokens and produces a C file that performs lexical analysis and identifies the tokens. Here we describe the functionality and construction of our scanner.

Functionality

Below is a list containing the different tokens that are identified by our Lexer program. It also gives a detailed description of how the different tokens are identified and how errors are detected if any.

Keywords

The keywords identified are: **int, long, short, long long, signed, unsigned, for, break, continue, if, else, return.**

Identifiers

1. **Recognition**: Identifiers are recognized based on the rules defined in the lexer. The regular expression used to recognize identifiers follows the pattern `(_|{letter})({letter}|{digit}|_){0,31}`, where `{letter}` and `{digit}` represent letters and digits, respectively.
2. **Validation**: Identifiers must adhere to certain rules, such as starting with a letter or underscore, followed by a combination of letters, digits, or underscores, with a maximum length of 32 characters. Any identifier that violates these rules is flagged as a lexical error.
3. **Storage**: Once an identifier is recognized and validated, it is added to the symbol table for further processing during parsing or semantic analysis stages.
4. **Error Handling**: If an identifier begins with a digit, it is considered invalid according to C language rules, and an appropriate error message is generated to indicate the lexical error.

Comments

Single line comments are identified by **//.*** regular expression. The multiline regex is identified as follows:

1. **Single-Line Comments:** Single-line comments in C, identified by `//`, are handled using the regular expression `//.*`. This expression matches any text following `//` until the end of the line, effectively identifying and ignoring single-line comments.
2. **Unterminated Comments:** If a multi-line comment is not terminated before the end of the file (`EOF`), it is detected by checking if the lexer matches the `<<EOF>>` pattern while still in the `<CMNT>` state. This indicates that the comment has not been properly terminated, and an error message along with the line number where the comment begins is displayed.

Strings

Strings are handled within the lexer() function, specifically in the lexer class defined within the code. Let's break down how strings are handled:

1. **Recognition**: Strings are recognized using regular expressions that match patterns within double quotes (").
2. **Error Handling**: Unterminated strings and escaped characters are handled to ensure accurate lexical analysis.
3. **Processing**: Once a string is recognized, it is processed accordingly, which may involve adding it to the constants table or flagging errors if necessary.
4. **Implementation Details**: Specific functions and logic within the lexer class are responsible for implementing string recognition and processing, ensuring robust lexical analysis of C programs.

Integer Constants

1. **Recognition**: The lexer identifies integer constants based on their pattern, typically consisting of one or more digits. For example, the integer constant `123` would be recognized by matching the pattern `[0-9]+`.
2. **Tokenization**: Once an integer constant is recognized, it is tokenized by creating a token with the appropriate type (e.g., "INTEGER_CONSTANT") and storing its value (e.g., `123`) along with its line number.
3. **Error Handling**: Error handling may involve detecting invalid integer constants, such as those containing non-numeric characters or exceeding the representable range for integers. If an invalid integer constant is encountered, an appropriate error message is generated and the lexical analysis may be halted or continued depending on the error recovery strategy.
4. **Integration with Parser**: The tokens representing integer constants, along with tokens for other lexical elements, are passed to the parser for syntactic analysis. The parser utilizes these tokens to recognize and construct the syntax tree of the input program.

Preprocessor Directives

1. **Recognition:**
   - Preprocessor directives start with a '#' symbol followed by the directive name.
   - We define rules in our lexer to recognize preprocessor directives based on this pattern.
2. **Tokenization:**
   - Once a preprocessor directive is recognized, we tokenize it as a separate token with its own token type.
   - For example, we may tokenize `#include`, `#define`, `#ifdef`, `#endif`, etc., each with its own token type.
3. **Error Handling:**
   - We handle errors related to preprocessor directives, such as invalid or unrecognized directives, by providing appropriate error messages.
   - For example, if a preprocessor directive is not recognized or contains a syntax error, we generate an error message indicating the issue.
4. **Processing:**
   - Depending on the design of our lexer and parser, we may choose to either process preprocessor directives within the lexer itself or pass them to the parser for further processing.
   - Processing may involve actions such as including header files, defining macros, conditional compilation, etc.
5. **Integration with Parser:**
   - If the parser needs to be aware of preprocessor directives, we ensure that the lexer tokenizes them appropriately and passes them to the parser.
   - The parser then interprets these tokens and performs the necessary actions based on the directives encountered

Test Cases

Test Case 1

```
Enter you statement below:
#include <iostream> using namespace std; void main () { if(a<b) {return i++;}else{i=0;}}
INPUT FILE:
 #include <iostream> using namespace std; void main () { if(a<b) {return i++;}else{i=0;}}
{'value': '#', 'LINE_NUMBERS': 1, 'TYPE': 'SHARP'}
{'value': 'include', 'LINE_NUMBERS': 1, 'TYPE': 'KEY_WORD'}
{'value': '<', 'LINE_NUMBERS': 1, 'TYPE': 'LT'}
{'value': 'iostream', 'LINE_NUMBERS': 1, 'TYPE': 'IDENTIFIER'}
{'value': '>', 'LINE_NUMBERS': 1, 'TYPE': 'GT'}
{'value': 'using', 'LINE_NUMBERS': 1, 'TYPE': 'KEY_WORD'}
{'value': 'namespace', 'LINE_NUMBERS': 1, 'TYPE': 'KEY_WORD'}
{'value': 'std', 'LINE_NUMBERS': 1, 'TYPE': 'IDENTIFIER'}
{'value': ';', 'LINE_NUMBERS': 1, 'TYPE': 'SEMICOLON'}
{'value': 'void', 'LINE_NUMBERS': 1, 'TYPE': 'DATATYPE'}
{'value': 'main', 'LINE_NUMBERS': 1, 'TYPE': 'IDENTIFIER'}
{'value': '(', 'LINE_NUMBERS': 1, 'TYPE': 'LL_BRACKET'}
{'value': ')', 'LINE_NUMBERS': 1, 'TYPE': 'RL_BRACKET'}
{'value': '{', 'LINE_NUMBERS': 1, 'TYPE': 'LB_BRACKET'}
{'value': 'if', 'LINE_NUMBERS': 1, 'TYPE': 'KEY_WORD'}
{'value': '(', 'LINE_NUMBERS': 1, 'TYPE': 'LL_BRACKET'}
{'value': 'a', 'LINE_NUMBERS': 1, 'TYPE': 'IDENTIFIER'}
{'value': '<', 'LINE_NUMBERS': 1, 'TYPE': 'LT'}
{'value': 'b', 'LINE_NUMBERS': 1, 'TYPE': 'IDENTIFIER'}
{'value': ')', 'LINE_NUMBERS': 1, 'TYPE': 'RL_BRACKET'}
{'value': '{', 'LINE_NUMBERS': 1, 'TYPE': 'LB_BRACKET'}
{'value': 'return', 'LINE_NUMBERS': 1, 'TYPE': 'KEY_WORD'}
{'value': 'i', 'LINE_NUMBERS': 1, 'TYPE': 'IDENTIFIER'}
{'value': '++', 'LINE_NUMBERS': 1, 'TYPE': 'SELF_PLUS'}
{'value': ';', 'LINE_NUMBERS': 1, 'TYPE': 'SEMICOLON'}
{'value': '}', 'LINE_NUMBERS': 1, 'TYPE': 'RB_BRACKET'}
{'value': 'else', 'LINE_NUMBERS': 1, 'TYPE': 'KEY_WORD'}
{'value': '{', 'LINE_NUMBERS': 1, 'TYPE': 'LB_BRACKET'}
{'value': 'i', 'LINE_NUMBERS': 1, 'TYPE': 'IDENTIFIER'}
{'value': '=', 'LINE_NUMBERS': 1, 'TYPE': 'ASSIGN'}
{'value': '0', 'LINE_NUMBERS': 1, 'TYPE': 'INTEGER_CONSTANT'}
{'value': ';', 'LINE_NUMBERS': 1, 'TYPE': 'SEMICOLON'}
{'value': '}', 'LINE_NUMBERS': 1, 'TYPE': 'RB_BRACKET'}
{'value': '}', 'LINE_NUMBERS': 1, 'TYPE': 'RB_BRACKET'}
{'value': '$', 'LINE_NUMBERS': 2, 'TYPE': 'EOF'}
SYNTAX IS CORRECT...
```

Test Case 2

```
Enter you statement below:
#include <iostream> using namespace std; void main () { if(a<b) {return i++;}else{i=0;}switch(a){case 'A': cin>>input;cout<<abcd;default: break;}}
INPUT FILE:
 #include <iostream> using namespace std; void main () { if(a<b) {return i++;}else{i=0;}switch(a){case 'A': cin>>input;cout<<abcd;default: break;}}
{'value': '#', 'LINE_NUMBERS': 1, 'TYPE': 'SHARP'}
{'value': 'include', 'LINE_NUMBERS': 1, 'TYPE': 'KEY_WORD'}
{'value': '<', 'LINE_NUMBERS': 1, 'TYPE': 'LT'}
{'value': 'iostream', 'LINE_NUMBERS': 1, 'TYPE': 'IDENTIFIER'}
{'value': '>', 'LINE_NUMBERS': 1, 'TYPE': 'GT'}
{'value': 'using', 'LINE_NUMBERS': 1, 'TYPE': 'KEY_WORD'}
{'value': 'namespace', 'LINE_NUMBERS': 1, 'TYPE': 'KEY_WORD'}
{'value': 'std', 'LINE_NUMBERS': 1, 'TYPE': 'IDENTIFIER'}
{'value': ';', 'LINE_NUMBERS': 1, 'TYPE': 'SEMICOLON'}
{'value': 'void', 'LINE_NUMBERS': 1, 'TYPE': 'DATATYPE'}
{'value': 'main', 'LINE_NUMBERS': 1, 'TYPE': 'IDENTIFIER'}
{'value': '(', 'LINE_NUMBERS': 1, 'TYPE': 'LL_BRACKET'}
{'value': ')', 'LINE_NUMBERS': 1, 'TYPE': 'RL_BRACKET'}
{'value': '{', 'LINE_NUMBERS': 1, 'TYPE': 'LB_BRACKET'}
{'value': 'if', 'LINE_NUMBERS': 1, 'TYPE': 'KEY_WORD'}
{'value': '(', 'LINE_NUMBERS': 1, 'TYPE': 'LL_BRACKET'}
{'value': 'a', 'LINE_NUMBERS': 1, 'TYPE': 'IDENTIFIER'}
{'value': '<', 'LINE_NUMBERS': 1, 'TYPE': 'LT'}
{'value': 'b', 'LINE_NUMBERS': 1, 'TYPE': 'IDENTIFIER'}
{'value': ')', 'LINE_NUMBERS': 1, 'TYPE': 'RL_BRACKET'}
{'value': '{', 'LINE_NUMBERS': 1, 'TYPE': 'LB_BRACKET'}
{'value': 'return', 'LINE_NUMBERS': 1, 'TYPE': 'KEY_WORD'}
{'value': 'i', 'LINE_NUMBERS': 1, 'TYPE': 'IDENTIFIER'}
{'value': '++', 'LINE_NUMBERS': 1, 'TYPE': 'SELF_PLUS'}
{'value': ';', 'LINE_NUMBERS': 1, 'TYPE': 'SEMICOLON'}
{'value': '}', 'LINE_NUMBERS': 1, 'TYPE': 'RB_BRACKET'}
{'value': 'else', 'LINE_NUMBERS': 1, 'TYPE': 'KEY_WORD'}
{'value': '{', 'LINE_NUMBERS': 1, 'TYPE': 'LB_BRACKET'}
{'value': 'i', 'LINE_NUMBERS': 1, 'TYPE': 'IDENTIFIER'}
{'value': '=', 'LINE_NUMBERS': 1, 'TYPE': 'ASSIGN'}
{'value': '0', 'LINE_NUMBERS': 1, 'TYPE': 'INTEGER_CONSTANT'}
{'value': ';', 'LINE_NUMBERS': 1, 'TYPE': 'SEMICOLON'}
{'value': '}', 'LINE_NUMBERS': 1, 'TYPE': 'RB_BRACKET'}
{'value': 'switch', 'LINE_NUMBERS': 1, 'TYPE': 'KEY_WORD'}
{'value': '(', 'LINE_NUMBERS': 1, 'TYPE': 'LL_BRACKET'}
{'value': 'a', 'LINE_NUMBERS': 1, 'TYPE': 'IDENTIFIER'}
{'value': ')', 'LINE_NUMBERS': 1, 'TYPE': 'RL_BRACKET'}
{'value': '{', 'LINE_NUMBERS': 1, 'TYPE': 'LB_BRACKET'}
{'value': 'case', 'LINE_NUMBERS': 1, 'TYPE': 'KEY_WORD'}
{'value': 'A', 'LINE_NUMBERS': 1, 'TYPE': 'CHAR_CONSTANTS'}
{'value': ':', 'LINE_NUMBERS': 1, 'TYPE': 'COLON'}
{'value': 'cin', 'LINE_NUMBERS': 1, 'TYPE': 'KEY_WORD'}
{'value': '>>', 'LINE_NUMBERS': 1, 'TYPE': 'R_SHIFT'}
{'value': 'input', 'LINE_NUMBERS': 1, 'TYPE': 'IDENTIFIER'}
{'value': ';', 'LINE_NUMBERS': 1, 'TYPE': 'SEMICOLON'}
{'value': 'cout', 'LINE_NUMBERS': 1, 'TYPE': 'KEY_WORD'}
{'value': '<<', 'LINE_NUMBERS': 1, 'TYPE': 'L_SHIFT'}
{'value': 'abcd', 'LINE_NUMBERS': 1, 'TYPE': 'IDENTIFIER'}
{'value': ';', 'LINE_NUMBERS': 1, 'TYPE': 'SEMICOLON'}
{'value': 'default', 'LINE_NUMBERS': 1, 'TYPE': 'KEY_WORD'}
{'value': ':', 'LINE_NUMBERS': 1, 'TYPE': 'COLON'}
{'value': 'break', 'LINE_NUMBERS': 1, 'TYPE': 'KEY_WORD'}
{'value': ';', 'LINE_NUMBERS': 1, 'TYPE': 'SEMICOLON'}
{'value': '}', 'LINE_NUMBERS': 1, 'TYPE': 'RB_BRACKET'}
{'value': '}', 'LINE_NUMBERS': 1, 'TYPE': 'RB_BRACKET'}
{'value': '$', 'LINE_NUMBERS': 2, 'TYPE': 'EOF'}
SYNTAX IS CORRECT...
```

Test Case 3

```
Enter you statement below:
#include <iostream> using namespace std; void main () { if(a<b) {return i++}else{i=0;}}
INPUT FILE:
 #include <iostream> using namespace std; void main () { if(a<b) {return i++}else{i=0;}}
{'value': '#', 'LINE_NUMBERS': 1, 'TYPE': 'SHARP'}
{'value': 'include', 'LINE_NUMBERS': 1, 'TYPE': 'KEY_WORD'}
{'value': '<', 'LINE_NUMBERS': 1, 'TYPE': 'LT'}
{'value': 'iostream', 'LINE_NUMBERS': 1, 'TYPE': 'IDENTIFIER'}
{'value': '>', 'LINE_NUMBERS': 1, 'TYPE': 'GT'}
{'value': 'using', 'LINE_NUMBERS': 1, 'TYPE': 'KEY_WORD'}
{'value': 'namespace', 'LINE_NUMBERS': 1, 'TYPE': 'KEY_WORD'}
{'value': 'std', 'LINE_NUMBERS': 1, 'TYPE': 'IDENTIFIER'}
{'value': ';', 'LINE_NUMBERS': 1, 'TYPE': 'SEMICOLON'}
{'value': 'void', 'LINE_NUMBERS': 1, 'TYPE': 'DATATYPE'}
{'value': 'main', 'LINE_NUMBERS': 1, 'TYPE': 'IDENTIFIER'}
{'value': '(', 'LINE_NUMBERS': 1, 'TYPE': 'LL_BRACKET'}
{'value': ')', 'LINE_NUMBERS': 1, 'TYPE': 'RL_BRACKET'}
{'value': '{', 'LINE_NUMBERS': 1, 'TYPE': 'LB_BRACKET'}
{'value': 'if', 'LINE_NUMBERS': 1, 'TYPE': 'KEY_WORD'}
{'value': '(', 'LINE_NUMBERS': 1, 'TYPE': 'LL_BRACKET'}
{'value': 'a', 'LINE_NUMBERS': 1, 'TYPE': 'IDENTIFIER'}
{'value': '<', 'LINE_NUMBERS': 1, 'TYPE': 'LT'}
{'value': 'b', 'LINE_NUMBERS': 1, 'TYPE': 'IDENTIFIER'}
{'value': ')', 'LINE_NUMBERS': 1, 'TYPE': 'RL_BRACKET'}
{'value': '{', 'LINE_NUMBERS': 1, 'TYPE': 'LB_BRACKET'}
{'value': 'return', 'LINE_NUMBERS': 1, 'TYPE': 'KEY_WORD'}
{'value': 'i', 'LINE_NUMBERS': 1, 'TYPE': 'IDENTIFIER'}
{'value': '++', 'LINE_NUMBERS': 1, 'TYPE': 'SELF_PLUS'}
{'value': '}', 'LINE_NUMBERS': 1, 'TYPE': 'RB_BRACKET'}
{'value': 'else', 'LINE_NUMBERS': 1, 'TYPE': 'KEY_WORD'}
{'value': '{', 'LINE_NUMBERS': 1, 'TYPE': 'LB_BRACKET'}
{'value': 'i', 'LINE_NUMBERS': 1, 'TYPE': 'IDENTIFIER'}
{'value': '=', 'LINE_NUMBERS': 1, 'TYPE': 'ASSIGN'}
{'value': '0', 'LINE_NUMBERS': 1, 'TYPE': 'INTEGER_CONSTANT'}
{'value': ';', 'LINE_NUMBERS': 1, 'TYPE': 'SEMICOLON'}
{'value': '}', 'LINE_NUMBERS': 1, 'TYPE': 'RB_BRACKET'}
{'value': '}', 'LINE_NUMBERS': 1, 'TYPE': 'RB_BRACKET'}
{'value': '$', 'LINE_NUMBERS': 2, 'TYPE': 'EOF'}
ERROR WITH SYNATX NEAR TOKKEN:  }  IN LINE NUMBER:  1
```

Test Case 4

```
Enter you statement below:
#include <iostream using namespace std; void main () { if(a<b) {return i++;}else{i=0;}switch(a){case 'A': cin>>input;cout<<abcd;default: break;}}
INPUT FILE:
 #include <iostream using namespace std; void main () { if(a<b) {return i++;}else{i=0;}switch(a){case 'A': cin>>input;cout<<abcd;default: break;}}
{'value': '#', 'LINE_NUMBERS': 1, 'TYPE': 'SHARP'}
{'value': 'include', 'LINE_NUMBERS': 1, 'TYPE': 'KEY_WORD'}
{'value': '<', 'LINE_NUMBERS': 1, 'TYPE': 'LT'}
{'value': 'iostream', 'LINE_NUMBERS': 1, 'TYPE': 'IDENTIFIER'}
{'value': 'using', 'LINE_NUMBERS': 1, 'TYPE': 'KEY_WORD'}
{'value': 'namespace', 'LINE_NUMBERS': 1, 'TYPE': 'KEY_WORD'}
{'value': 'std', 'LINE_NUMBERS': 1, 'TYPE': 'IDENTIFIER'}
{'value': ';', 'LINE_NUMBERS': 1, 'TYPE': 'SEMICOLON'}
{'value': 'void', 'LINE_NUMBERS': 1, 'TYPE': 'DATATYPE'}
{'value': 'main', 'LINE_NUMBERS': 1, 'TYPE': 'IDENTIFIER'}
{'value': '(', 'LINE_NUMBERS': 1, 'TYPE': 'LL_BRACKET'}
{'value': ')', 'LINE_NUMBERS': 1, 'TYPE': 'RL_BRACKET'}
{'value': '{', 'LINE_NUMBERS': 1, 'TYPE': 'LB_BRACKET'}
{'value': 'if', 'LINE_NUMBERS': 1, 'TYPE': 'KEY_WORD'}
{'value': '(', 'LINE_NUMBERS': 1, 'TYPE': 'LL_BRACKET'}
{'value': 'a', 'LINE_NUMBERS': 1, 'TYPE': 'IDENTIFIER'}
{'value': '<', 'LINE_NUMBERS': 1, 'TYPE': 'LT'}
{'value': 'b', 'LINE_NUMBERS': 1, 'TYPE': 'IDENTIFIER'}
{'value': ')', 'LINE_NUMBERS': 1, 'TYPE': 'RL_BRACKET'}
{'value': '{', 'LINE_NUMBERS': 1, 'TYPE': 'LB_BRACKET'}
{'value': 'return', 'LINE_NUMBERS': 1, 'TYPE': 'KEY_WORD'}
{'value': 'i', 'LINE_NUMBERS': 1, 'TYPE': 'IDENTIFIER'}
{'value': '++', 'LINE_NUMBERS': 1, 'TYPE': 'SELF_PLUS'}
{'value': ';', 'LINE_NUMBERS': 1, 'TYPE': 'SEMICOLON'}
{'value': '}', 'LINE_NUMBERS': 1, 'TYPE': 'RB_BRACKET'}
{'value': 'else', 'LINE_NUMBERS': 1, 'TYPE': 'KEY_WORD'}
{'value': '{', 'LINE_NUMBERS': 1, 'TYPE': 'LB_BRACKET'}
{'value': 'i', 'LINE_NUMBERS': 1, 'TYPE': 'IDENTIFIER'}
{'value': '=', 'LINE_NUMBERS': 1, 'TYPE': 'ASSIGN'}
{'value': '0', 'LINE_NUMBERS': 1, 'TYPE': 'INTEGER_CONSTANT'}
{'value': ';', 'LINE_NUMBERS': 1, 'TYPE': 'SEMICOLON'}
{'value': '}', 'LINE_NUMBERS': 1, 'TYPE': 'RB_BRACKET'}
{'value': 'switch', 'LINE_NUMBERS': 1, 'TYPE': 'KEY_WORD'}
{'value': '(', 'LINE_NUMBERS': 1, 'TYPE': 'LL_BRACKET'}
{'value': 'a', 'LINE_NUMBERS': 1, 'TYPE': 'IDENTIFIER'}
{'value': ')', 'LINE_NUMBERS': 1, 'TYPE': 'RL_BRACKET'}
{'value': '{', 'LINE_NUMBERS': 1, 'TYPE': 'LB_BRACKET'}
{'value': 'case', 'LINE_NUMBERS': 1, 'TYPE': 'KEY_WORD'}
{'value': 'A', 'LINE_NUMBERS': 1, 'TYPE': 'CHAR_CONSTANTS'}
{'value': ':', 'LINE_NUMBERS': 1, 'TYPE': 'COLON'}
{'value': 'cin', 'LINE_NUMBERS': 1, 'TYPE': 'KEY_WORD'}
{'value': '>>', 'LINE_NUMBERS': 1, 'TYPE': 'R_SHIFT'}
{'value': 'input', 'LINE_NUMBERS': 1, 'TYPE': 'IDENTIFIER'}
{'value': ';', 'LINE_NUMBERS': 1, 'TYPE': 'SEMICOLON'}
{'value': 'cout', 'LINE_NUMBERS': 1, 'TYPE': 'KEY_WORD'}
{'value': '<<', 'LINE_NUMBERS': 1, 'TYPE': 'L_SHIFT'}
{'value': 'abcd', 'LINE_NUMBERS': 1, 'TYPE': 'IDENTIFIER'}
{'value': ';', 'LINE_NUMBERS': 1, 'TYPE': 'SEMICOLON'}
{'value': 'default', 'LINE_NUMBERS': 1, 'TYPE': 'KEY_WORD'}
{'value': ':', 'LINE_NUMBERS': 1, 'TYPE': 'COLON'}
{'value': 'break', 'LINE_NUMBERS': 1, 'TYPE': 'KEY_WORD'}
{'value': ';', 'LINE_NUMBERS': 1, 'TYPE': 'SEMICOLON'}
{'value': '}', 'LINE_NUMBERS': 1, 'TYPE': 'RB_BRACKET'}
{'value': '}', 'LINE_NUMBERS': 1, 'TYPE': 'RB_BRACKET'}
{'value': '$', 'LINE_NUMBERS': 2, 'TYPE': 'EOF'}
ERROR WITH SYNATX NEAR TOKKEN:  using  IN LINE NUMBER:  1
```

Conclusion

In conclusion, the development of our three-part code comprising a Lexer, Parser, and Main Program represents a significant milestone in facilitating the lexical and syntactic analysis of C programming language constructs. The Lexer, designed to tokenize the input source code, adeptly identifies various lexical elements including keywords, identifiers, literals, and preprocessor directives. Its robust error handling ensures the integrity of the token stream, laying a solid foundation for subsequent analysis.

Complementing the Lexer, our Parser seamlessly integrates with the token stream to perform syntactic analysis, adhering to the grammar rules of the C programming language. By constructing abstract syntax trees and executing semantic actions, the Parser enables comprehensive understanding and validation of the input source code's structure and semantics. Moreover, the Main Program orchestrates the interaction between the Lexer and Parser, streamlining the analysis process and providing a user-friendly interface for source code analysis.

Through meticulous organization and modular design, our three-part code not only facilitates efficient lexical and syntactic analysis but also fosters code maintainability and extensibility. By adhering to best practices in software engineering, including modularity, separation of concerns, and robust error handling, our codebase stands as a testament to our commitment to quality and reliability.

In the ever-evolving landscape of programming language analysis, our three-part code serves as a versatile tool for developers, educators, and researchers alike, empowering them to delve deeper into the intricacies of C programming language constructs with confidence and precision. As we continue to refine and expand our codebase, we remain dedicated to pushing the boundaries of lexical and syntactic analysis, driving innovation and excellence in software development.