Department of Computer Science & Information Technology

Bachelor of Science in Computer Science

Course Title: Theory of Programming Language

Course Code: CT-367

ASSIGNMENT TITLE:
DESIGN AND DEVELOPMENT OF A LEXICAL ANALYZER

SUBMITTED TO: SIR JASIM

Submitted By:
Yumna Irfan (CT-22004)
Yasha Ali (CT-22010)
Sara Razeen (CT-22049)
Submission Date: 5/5/2025

Complex Computing Problem Assessment Rubrics

Course Code: CT-367			**Course Title: Theory of Programming Language**
Criteria and Scales			
Excellent (3)	**Good (2)**	**Average (1)**	**Poor (0)**
Criterion 1: Understanding the Problem: How well the problem statement is understood by the student			
Understands the problem clearly and clearly identifies the underlying issues and functionalities.	Adequately understands the problem and identifies the underlying issues and functionalities.	Inadequately defines the problem and identifies the underlying issues and functionalities.	Fails to define the problem adequately and does not identify the underlying issues and functionalities.
Criterion 2: Research: The amount of research that is used in solving the problem			
Contains all the information needed for solving the problem	Good research, leading to a successful solution	Mediocre research which may or may not lead to an adequate solution	No apparent research
Criterion 3: Code: How complete the code is along with the assumptions and selected functionalities			
Complete Code according to the according to the selected functionalities of the given case with clear assumptions	Incomplete Code according to the selected functionalities of the given case with clear assumptions	Incomplete Code according to the selected functionalities of the given case with unclear assumptions	Wrong code and naming conventions
Criterion 4: Report: How thorough and well organized is the solution			
All the necessary information clearly organized for easy use in solving the problem	Good information organized well that could lead to a good solution	Mediocre information which may or may not lead to a solution	No report provided

Total Marks:_____

Teacher's Signature:_____

1. Introduction:

Lexical analysis is a fundamental step in the compilation process, responsible for reading the source code and breaking it into tokens. the smallest meaningful elements of the language. A lexical analyzer ensures that the source code is properly segmented for subsequent syntax and semantic analysis phases.

This project focuses on constructing a basic lexical analyzer for C++ using Flex and Bison. Flex (Fast Lexical Analyzer) generates code for scanning the input based on regular expression rules, while Bison can be used to define grammar for parsing sequences of tokens. The project emphasizes the tokenization phase, providing a strong foundation for understanding compiler construction.

2. Background:

Compilers rely heavily on lexical analyzers to simplify their input by removing irrelevant characters (such as whitespace and comments) and grouping character sequences into tokens. Traditionally, lexical analyzers were written manually; however, tools like Flex automate this process by allowing developers to specify patterns that correspond to token types.

Flex is widely used for building scanners due to its speed and flexibility. It can recognize complex patterns using regular expressions and generate highly efficient C code. Bison, a parser generator, complements Flex by handling the syntactic organization of tokens according to a specified grammar.

3. Choice of Language:

C++ is a general-purpose programming language developed by **Bjarne Stroustrup** at Bell Laboratories in the early 1980s. Initially called "**C with Classes**," C++ extended the C language by introducing object-oriented programming features while maintaining high performance and low-level control.

The first official version of C++ was released in **1985**, and since then it has evolved significantly through multiple standards such as C++98, C++11, C++17, and beyond. Today, C++ is widely used for building system software, compilers, game engines, and real-time applications due to its speed, efficiency, and flexibility.

In this project, C++ was selected as the target language for lexical analysis because of its complexity and rich set of features. Recognizing C++ tokens presents an interesting challenge due to the variety of constructs and syntax rules, making it an ideal choice for demonstrating the capabilities of Flex and Bison in handling real-world programming languages.

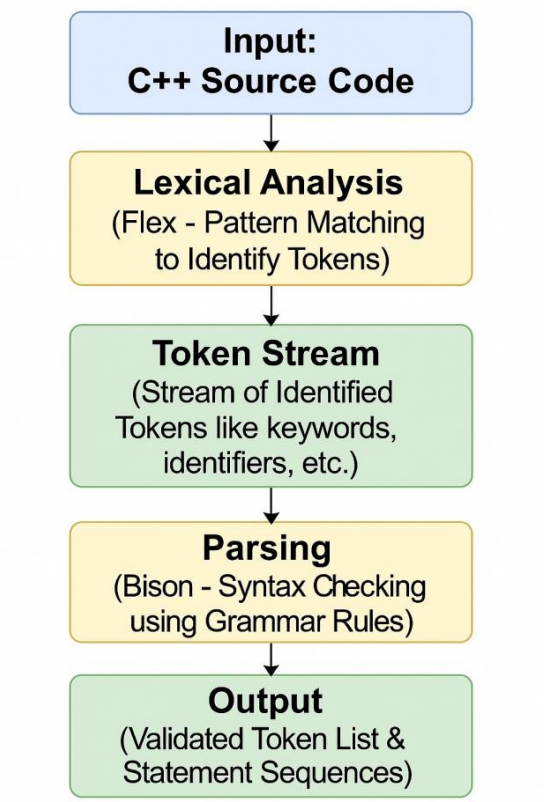
4. Tools and Technologies:

- Flex: A tool for generating lexical analyzers based on regular expression patterns.
- Bison: A parser gnerator that works alongside Flex to define grammar rules (optional usage in this project).
- C++: The target programming language for tokenization and lexical analysis.
- Windows Environment: The operating system used for development and testing.
- GCC Compiler (via MinGW or similar toolchain): Used to compile the generated scanner and parser code on Windows.

5. System Workflow:

The system follows a simple, structured flow to process C++ source code and generate tokens:

1. **Input**: The system accepts raw C++ source code as input.
2. **Lexical Analysis**: The Flex scanner processes the input code, matching patterns to identify tokens such as keywords, identifiers, constants, and operators etc.
3. **Token Stream**: As Flex identifies each token, it generates a stream of tokens that represent the meaningful components of the source code.
4. **Parsing**: Bison further processes the token stream to ensure basic syntactic correctness, checking for valid sequences based on predefined grammar rules.
5. **Output**: The system saves a list of recognized tokens in a file, while the terminal displays whether the parsed statements are syntactically valid or if any errors were found.



6. Selected C++ Functionalities Supported in the System:

This section outlines the various C++ language features that the system is capable of processing through its integrated lexical and syntactic analysis components. These capabilities span from fundamental language constructs to more advanced programming paradigms.

1. **String Literals:** The system accurately recognizes and processes double-quoted strings such as "hello" or "C++ string".
   Sample Flex Rule: \"([^\\\"]|\\.)*\" { yylval.str = strdup(yytext); return STRING_LITERAL; }

2. **Character Literals:** Handles individual character inputs like 'a', '\n', or '%'.
   Sample Flex Rule: \'([^\\\']|\\.)\' { yylval.ch = yytext[1]; return CHAR_LITERAL; }

3. **Basic Input/Output:**
   a. Standard input using cin.
   b. Standard output using cout
   c. Line termination using endl

4. **Data Types:**

   a. Primitive types: int, float, double, char, bool, string
   b. const qualified types
   c. Pointer types (e.g., int*, char*)

5. **Declarations:**
   a. Variable declarations with or without initialization
   b. One-dimensional array declarations
   c. Comma-separated multiple declarations

6. **Expressions:**

   - **Arithmetic:** +, -, *, /, %

   - **Comparison:** ==, !=, <, >, <=, >=
   - **Logical:** &&, ||, !
   - **Bitwise:** &, |, ^, <<, >>
   - **Assignment:** =, +=, -=, etc.
   - **Unary Operators:** ++, --, *, &
   - **Ternary Operator:** ?:
   - **Function Calls**

7. **Control Structures:**

   a. Conditional statements: if, else

   b. Switch-case constructs: switch, case, default
   c. Looping constructs: for, while, do-while
   d. Flow control: break, continue, return

8. **Functions:**

   a. Function Declarations: Including return type, name, and parameter list

   b. Function Definitions: Full implementation of the function body
   c. Recognition of the main function: int main()

9. **Object-Oriented Programming:**

   a. Accessing class members: obj.member, obj.method()

   b. Support for the this pointer
   c. Class definitions (with assumed underlying grammar support)

10. **Exception Handling:**

    - Structured exception mechanisms: try, catch, throw

11. **Preprocessor Directives:**

    - Macros and conditional compilation: #define, #ifdef, #ifndef, #endif, #else, #undef

- File inclusion: #include

12. Namespaces:

- Namespace usage: using namespace std;

These functionalities are enabled through the collaborative design of the system's scanner and parser components. Lexical analysis (handled by Flex) identifies token patterns, while syntactic analysis (performed by Bison) ensures the structural correctness of C++ statements. Together, they provide an environment for interpreting and validating C++ source code.

7. Regular Expressions / Rules:

Token Type	Pattern/Regex	Action
Whitespace (WS)	[\t]+	Skip whitespace
Newline	\n	Increment line number
Header Include (C++ Directive)	"#include"[\t]*"<"[^>]+">", "#include"[\t]*\"[^"]+\"	Return HEADER_INCLUDE token
Preprocessor Define	"#define"[\t]+[a-zA-Z_][a-zA-Z0-9_]*[\t]*(.*)	Return PREPROCESSOR_DEFINE token
Preprocessor Ifdef	"#ifdef"[\t]+[a-zA-Z_][a-zA-Z0-9_]*	Return PREPROCESSOR_IFDEF token
Preprocessor Ifndef	"#ifndef"[\t]+[a-zA-Z_][a-zA-Z0-9_]*	Return PREPROCESSOR_IFNDEF token
Preprocessor Endif	"#endif"	Return PREPROCESSOR_ENDIF token
Preprocessor Else	"#else"	Return PREPROCESSOR_ELSE token
Preprocessor Undef	"#undef"[\t]+[a-zA-Z_][a-zA-Z0-9_]*	Return PREPROCESSOR_UNDEF token

Keywords (e.g., using, namespace, etc.)	"using", "namespace", "std", "cout", "endl", ...	Return respective keyword token
Number (Floating)	{DIGIT}+"."{DIGIT}+	Return FLOAT_NUM token
Number (Integer)	{DIGIT}+	Return NUMBER token
Character Literal	{CHAR_CONST}	Return CHAR_LITERAL token
String Literal	{STRING_CONST}	Return STRING_LITERAL token
Identifier (Variable/Function)	{ID_START}{ID_CHAR}*	Return ID token
Single-Line Comment	"//".*	Ignore single-line comment
Multi-Line Comment	`"/"([^]	*+[^/])"*/"`
Operators	"~", "==", "=", "!=", ...	Return respective operator token
Punctuation	";", ",", "\"", "{", "}", "(", ")", ...	Return respective punctuation token
Unknown Characters	.	Print unknown character error message

8. Source Code (C++):

- Lex Code: https://github.com/Sara-Razeen/C-_Compiler/blob/main/lexer.l
- Bison Code: https://github.com/Sara-Razeen/C-_Compiler/blob/main/parser.y

9. Output:

Section A: Parsing a C++ File with No Errors

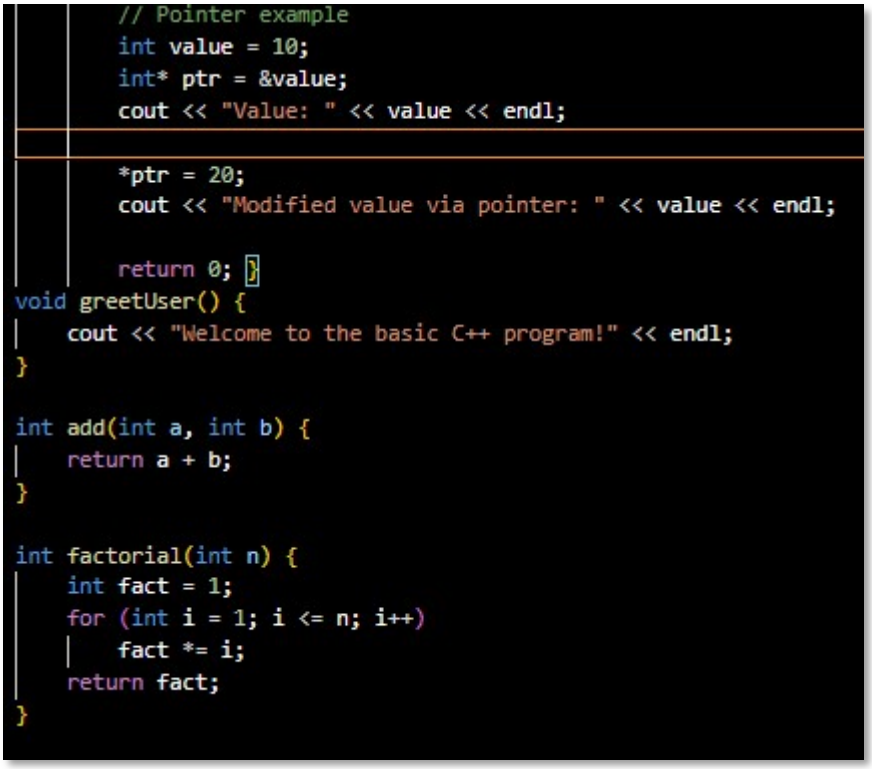
1. Input C++ File (No Errors):

Screenshots:

```cpp
#include <iostream>
#include <string>
using namespace std;
#define MAX 100

// Class definition with encapsulation
class Person {
private:
    string name;
public:
    void setName(string n) {
        name = n;
        if (!n.empty()) {
            name = n;
        } else {
            cout << "Error: Name cannot be empty" << endl;
        }
    }
    string getName() {
        return name;
        cout<<"Output statement"<<endl;
    }
    void inputName() {
        cout << "Enter your name: ";
        cin >> name;
    }
    void greet() {
        cout << "Hello, " << name << "!" << endl;
    }
    ~Person(){}
};

// Derived class using inheritance
class Student : public Person {
private:
    int rollNumber;

public:
    void setRollNumber(int r) {
```

```
class MyClass {
    public:
        virtual void display();  // Matches: VIRTUAL type_specifier IDENTIFIER '(' parameter_list_opt ')' ';'
    };

class MyClass {
    public:
        virtual void show() {
            cout<<"test"<<endl;
        }  // Matches: VIRTUAL type_specifier IDENTIFIER '(' parameter_list_opt ')' compound_stmt
        };

// Function declarations
void greetUser();
int add(int a, int b);
int factorial(int n);

int main() {
    Student user;

    int num1, num2;
    greetUser(); // Call void function

    user.inputName();  // Inherited input
    user.greet();      // Inherited greeting

    // Demonstrate Student-specific functionality
    int roll;
    cout << "Enter your roll number: ";
    cin >> roll;
    user.setRollNumber(roll);
    user.showDetails();
```

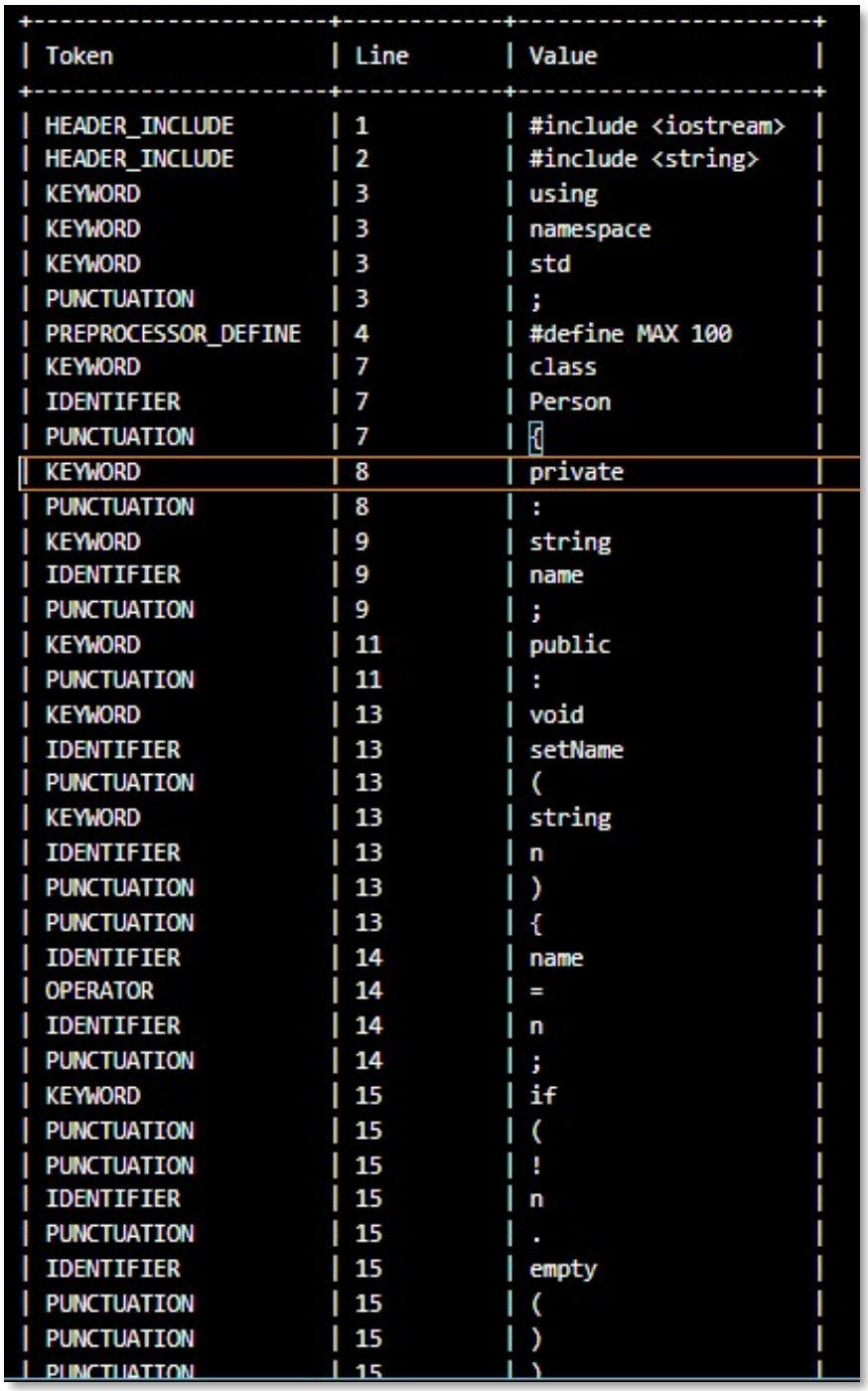
```
    cout << "Enter your roll number: ";
    cin >> roll;
    user.setRollNumber(roll);
    user.showDetails();

    /* Simple addition
    of num1 and num2*/
    cout << "Enter two integers to add: ";
    cin >> num1 >> num2;
    int result = add(num1, num2);
    cout << "Sum = " << result << endl;

    // Factorial using function
    int n;
    cout << "Enter a number to calculate its factorial: ";
    cin >> n;
    cout << "Factorial of " << n << " is " << factorial(n) << endl;

    // Array and loop
    int numbers[5];
    cout << "Enter 5 numbers: ";
    for (int i = 0; i < 5; i++) {
        cin >> numbers[i];
    }

    //Implementing try block
    try {
        cout << "Trying risky operation..." << endl;
        throw 5;
    } catch (int e) {
        cout << "Caught exception: " << e << endl;
    }
    cout << "You entered: ";
    for (int i = 0; i < 5; i++) {
        cout << numbers[i] << " ";
    }
        // Pointer example
        int value = 10;
```



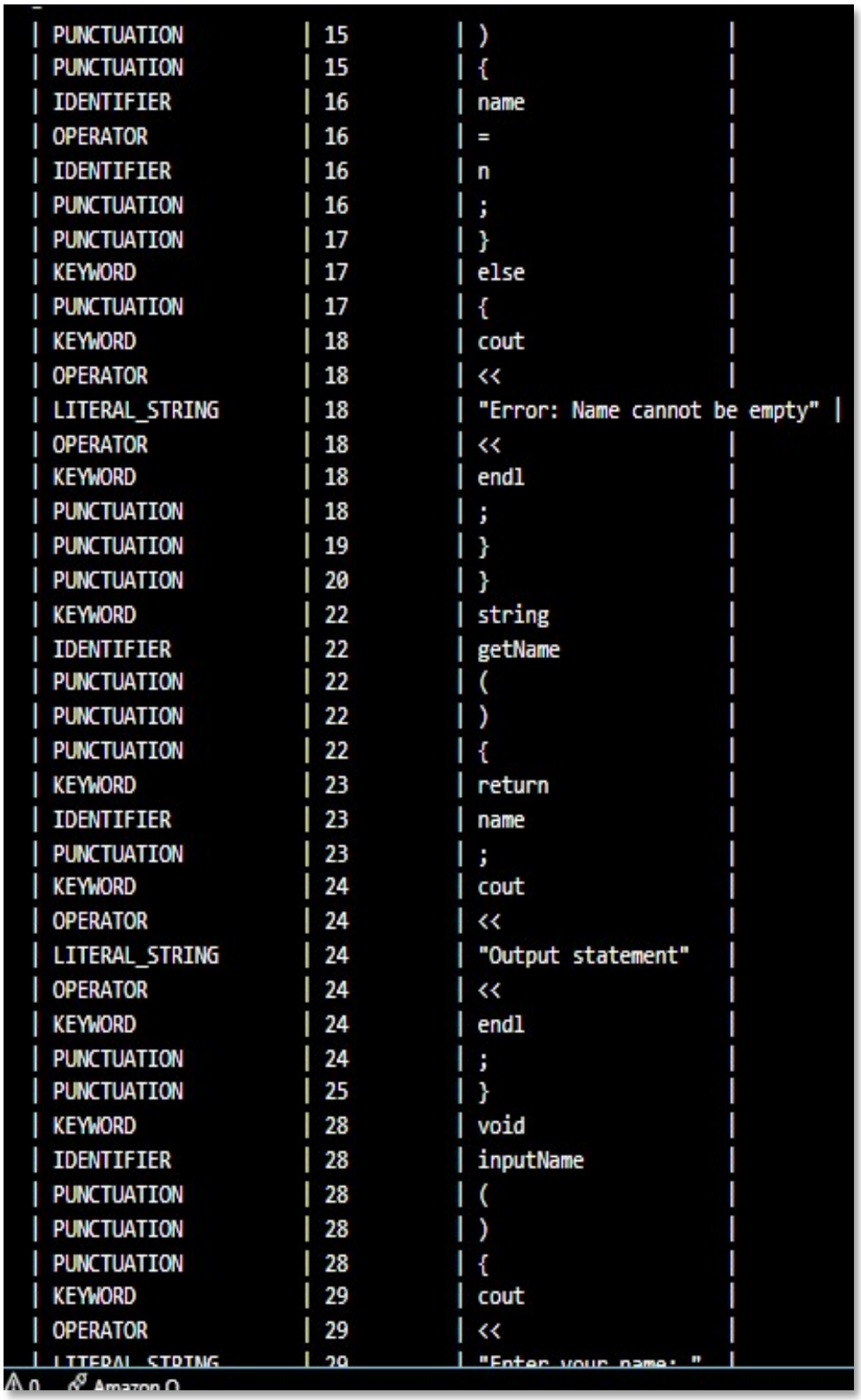
```
    // Pointer example
    int value = 10;
    int* ptr = &value;
    cout << "Value: " << value << endl;

    *ptr = 20;
    cout << "Modified value via pointer: " << value << endl;

    return 0; }
void greetUser() {
    cout << "Welcome to the basic C++ program!" << endl;
}

int add(int a, int b) {
    return a + b;
}

int factorial(int n) {
    int fact = 1;
    for (int i = 1; i <= n; i++)
        fact *= i;
    return fact;
}
```

Description: This is the original C++ source code submitted to the lexical analyzer and parser. The code is syntactically correct, containing valid declarations, control structures, and properly terminated statements. It serves as the input for tokenization and parsing.

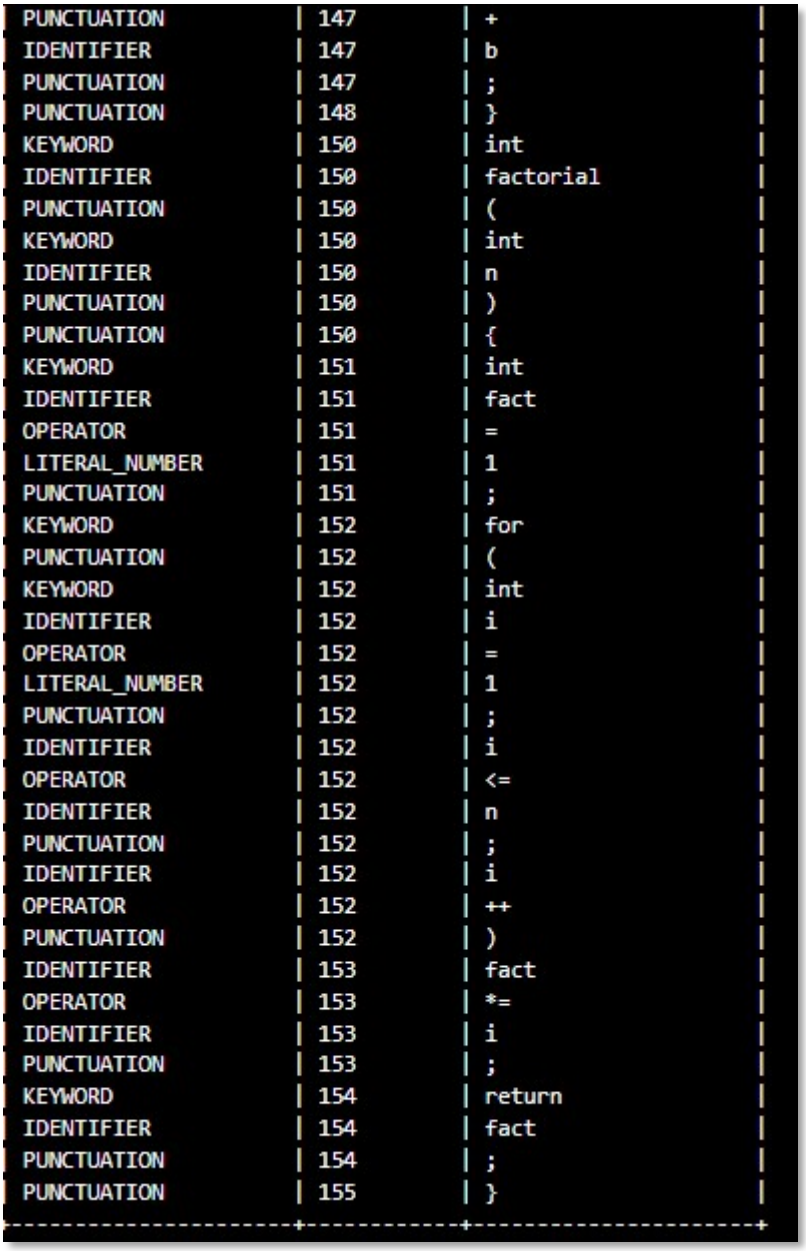
2. **Tokens Table (No Errors):**
   Screenshots:



Token	Line	Value
HEADER_INCLUDE	1	#include <iostream>
HEADER_INCLUDE	2	#include <string>
KEYWORD	3	using
KEYWORD	3	namespace
KEYWORD	3	std
PUNCTUATION	3	;
PREPROCESSOR_DEFINE	4	#define MAX 100
KEYWORD	7	class
IDENTIFIER	7	Person
PUNCTUATION	7	{
KEYWORD	8	private
PUNCTUATION	8	:
KEYWORD	9	string
IDENTIFIER	9	name
PUNCTUATION	9	;
KEYWORD	11	public
PUNCTUATION	11	:
KEYWORD	13	void
IDENTIFIER	13	setName
PUNCTUATION	13	(
KEYWORD	13	string
IDENTIFIER	13	n
PUNCTUATION	13	)
PUNCTUATION	13	{
IDENTIFIER	14	name
OPERATOR	14	=
IDENTIFIER	14	n
PUNCTUATION	14	;
KEYWORD	15	if
PUNCTUATION	15	(
PUNCTUATION	15	!
IDENTIFIER	15	n
PUNCTUATION	15	.
IDENTIFIER	15	empty
PUNCTUATION	15	(
PUNCTUATION	15	)
PUNCTUATION	15	)



PUNCTUATION	15	)
PUNCTUATION	15	{
IDENTIFIER	16	name
OPERATOR	16	=
IDENTIFIER	16	n
PUNCTUATION	16	;
PUNCTUATION	17	}
KEYWORD	17	else
PUNCTUATION	17	{
KEYWORD	18	cout
OPERATOR	18	<<
LITERAL_STRING	18	"Error: Name cannot be empty"
OPERATOR	18	<<
KEYWORD	18	endl
PUNCTUATION	18	;
PUNCTUATION	19	}
PUNCTUATION	20	}
KEYWORD	22	string
IDENTIFIER	22	getName
PUNCTUATION	22	(
PUNCTUATION	22	)
PUNCTUATION	22	{
KEYWORD	23	return
IDENTIFIER	23	name
PUNCTUATION	23	;
KEYWORD	24	cout
OPERATOR	24	<<
LITERAL_STRING	24	"Output statement"
OPERATOR	24	<<
KEYWORD	24	endl
PUNCTUATION	24	;
PUNCTUATION	25	}
KEYWORD	28	void
IDENTIFIER	28	inputName
PUNCTUATION	28	(
PUNCTUATION	28	)
PUNCTUATION	28	{
KEYWORD	29	cout
OPERATOR	29	<<
LITERAL_STRING	29	"Enter your name: "



PUNCTUATION	147	+
IDENTIFIER	147	b
PUNCTUATION	147	;
PUNCTUATION	148	}
KEYWORD	150	int
IDENTIFIER	150	factorial
PUNCTUATION	150	(
KEYWORD	150	int
IDENTIFIER	150	n
PUNCTUATION	150	)
PUNCTUATION	150	{
KEYWORD	151	int
IDENTIFIER	151	fact
OPERATOR	151	=
LITERAL_NUMBER	151	1
PUNCTUATION	151	;
KEYWORD	152	for
PUNCTUATION	152	(
KEYWORD	152	int
IDENTIFIER	152	i
OPERATOR	152	=
LITERAL_NUMBER	152	1
PUNCTUATION	152	;
IDENTIFIER	152	i
OPERATOR	152	<=
IDENTIFIER	152	n
PUNCTUATION	152	;
IDENTIFIER	152	i
OPERATOR	152	++
PUNCTUATION	152	)
IDENTIFIER	153	fact
OPERATOR	153	*=
IDENTIFIER	153	i
PUNCTUATION	153	;
KEYWORD	154	return
IDENTIFIER	154	fact
PUNCTUATION	154	;
PUNCTUATION	155	}

Description: The table below contains the list of tokens identified in the input code. Each token is categorized based on its type, and the corresponding lexeme is shown. This validates that the lexical analyzer correctly recognized all tokens.

3. **Terminal Output (No Errors):**
   Screenshots:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  CODE REFERENCE LOG

PS C:\Users\sara\Desktop\COMPILER> win_bison -d -Wno-other -Wno-conflicts-sr -Wno-conflicts-rr parser.y
PS C:\Users\sara\Desktop\COMPILER> win_flex lexer.l
PS C:\Users\sara\Desktop\COMPILER> g++ parser.tab.c lex.yy.c -o parser.exe -mconsole
PS C:\Users\sara\Desktop\COMPILER> ./parser.exe input.cpp
PS C:\Users\sara\Desktop\COMPILER>
```

Description: This output confirms that the input file was parsed successfully. All tokens were identified, and no syntax errors were detected. The parser completed its execution and accepted the input.

Section B: Parsing a C++ File Containing Errors

This section presents how a C++ file with different types of syntax errors is parsed. Each sub-section includes the input source file, the resulting tokens (if any), and the terminal output.

B.1: Cout Statement Error

1. **Input C++ File (With Error):**
   Screenshot:

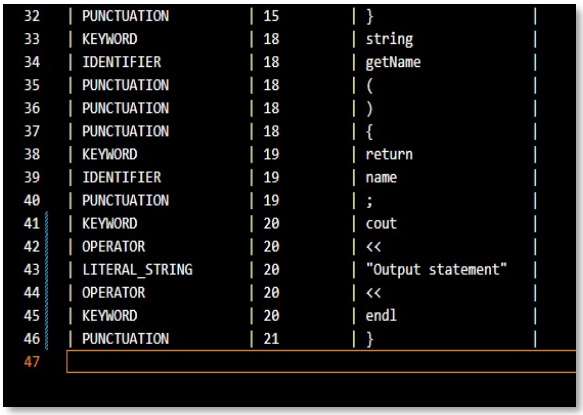
```
    cout<<"Output statement"<<endl
}
```

Description: This version of the source code contains an issue in the cout statement, which interrupts proper parsing.

2. **Tokens Table (With Error)**
   Screenshot:

token_table.txt

Token	Line	Value
HEADER_INCLUDE	1	#include <iostream>
HEADER_INCLUDE	2	#include <string>
KEYWORD	3	using
KEYWORD	3	namespace
KEYWORD	3	std
PUNCTUATION	3	;
PREPROCESSOR_DEFINE	4	#define MAX 100
KEYWORD	7	class
IDENTIFIER	7	Person
PUNCTUATION	7	{
KEYWORD	8	private
PUNCTUATION	8	:
KEYWORD	9	string
IDENTIFIER	9	name
PUNCTUATION	9	;
KEYWORD	11	public
PUNCTUATION	11	:
KEYWORD	13	void
IDENTIFIER	13	setName
PUNCTUATION	13	(
KEYWORD	13	string
IDENTIFIER	13	n
PUNCTUATION	13	)

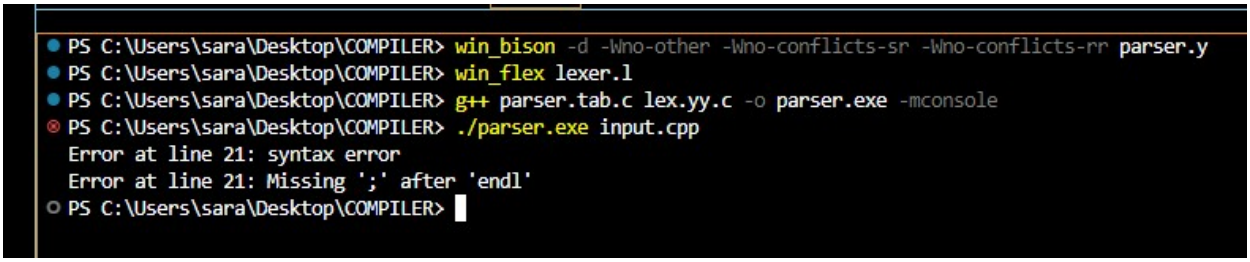


32	PUNCTUATION	15	}
33	KEYWORD	18	string
34	IDENTIFIER	18	getName
35	PUNCTUATION	18	(
36	PUNCTUATION	18	)
37	PUNCTUATION	18	{
38	KEYWORD	19	return
39	IDENTIFIER	19	name
40	PUNCTUATION	19	;
41	KEYWORD	20	cout
42	OPERATOR	20	<<
43	LITERAL_STRING	20	"Output statement"
44	OPERATOR	20	<<
45	KEYWORD	20	endl
46	PUNCTUATION	21	}
47			

Description: This token table lists all valid tokens identified before the parser stopped due to the output statement issue.

3. Terminal Output (With Error):

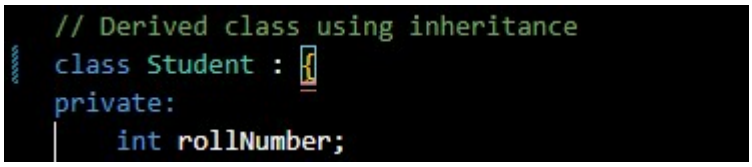
Screenshot:



Description: The parser identifies a syntax error in the output section and halts further processing, displaying an error message.

B.2: Inheritance Declaration Error

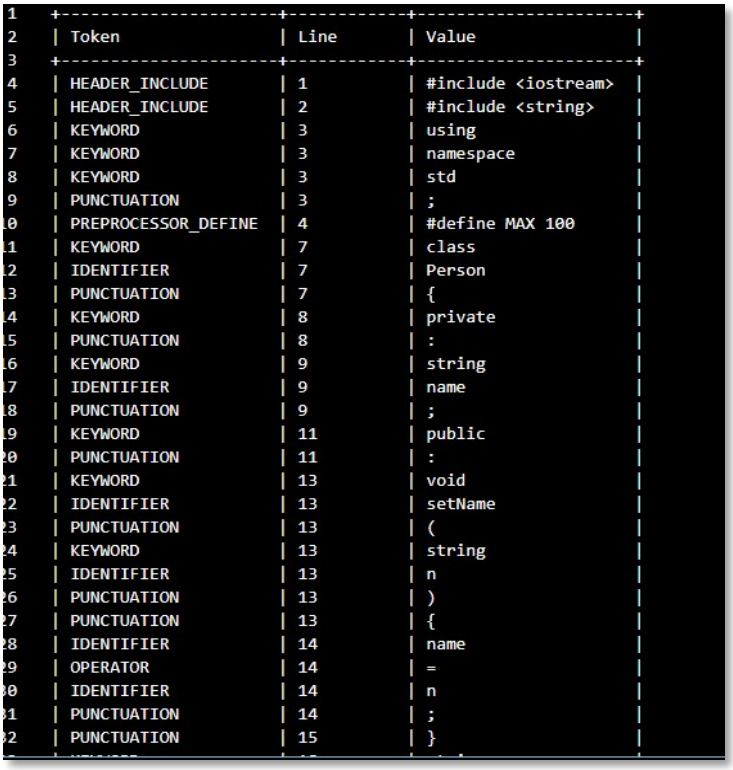
1. **Input C++ File (With Error)];**
   Screenshot



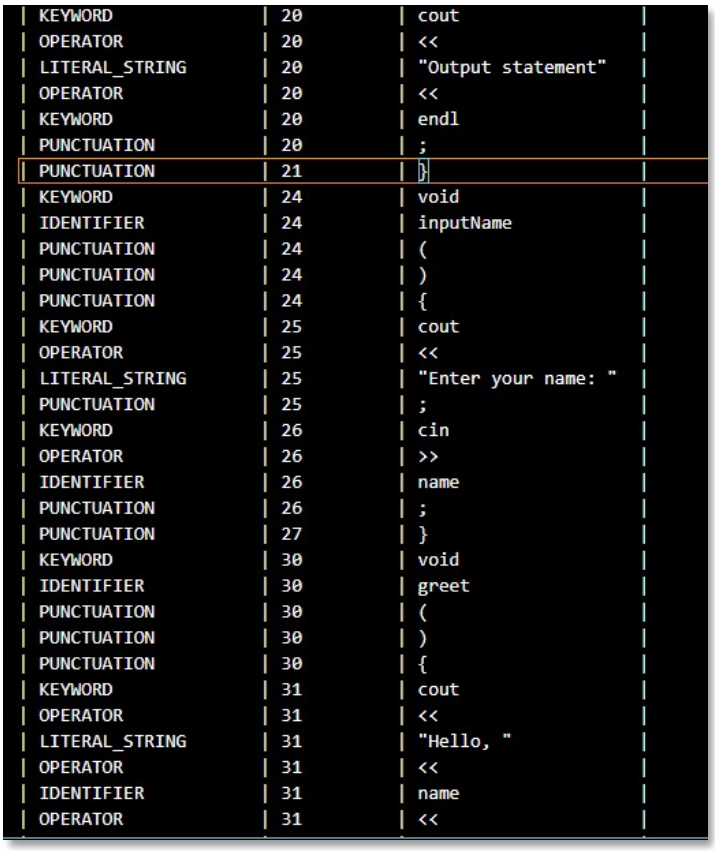
```
// Derived class using inheritance
class Student : {
private:
    int rollNumber;
```

Description: The inheritance syntax in the class definition is incorrect, affecting how the parser interprets the structure.

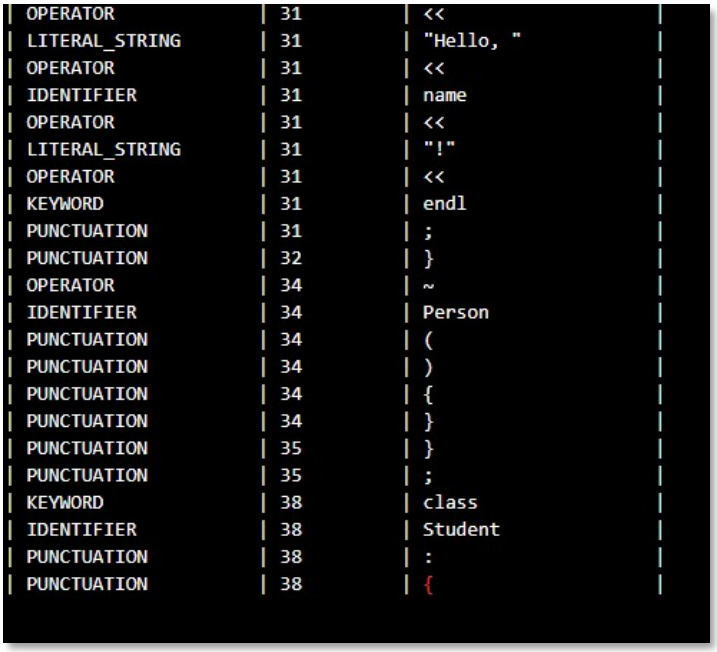
2. **Tokens Table (With Error):**
   Screenshot



Token	Line	Value
HEADER_INCLUDE	1	#include <iostream>
HEADER_INCLUDE	2	#include <string>
KEYWORD	3	using
KEYWORD	3	namespace
KEYWORD	3	std
PUNCTUATION	3	;
PREPROCESSOR_DEFINE	4	#define MAX 100
KEYWORD	7	class
IDENTIFIER	7	Person
PUNCTUATION	7	{
KEYWORD	8	private
PUNCTUATION	8	:
KEYWORD	9	string
IDENTIFIER	9	name
PUNCTUATION	9	;
KEYWORD	11	public
PUNCTUATION	11	:
KEYWORD	13	void
IDENTIFIER	13	setName
PUNCTUATION	13	(
KEYWORD	13	string
IDENTIFIER	13	n
PUNCTUATION	13	)
PUNCTUATION	13	{
IDENTIFIER	14	name
OPERATOR	14	=
IDENTIFIER	14	n
PUNCTUATION	14	;
PUNCTUATION	15	}

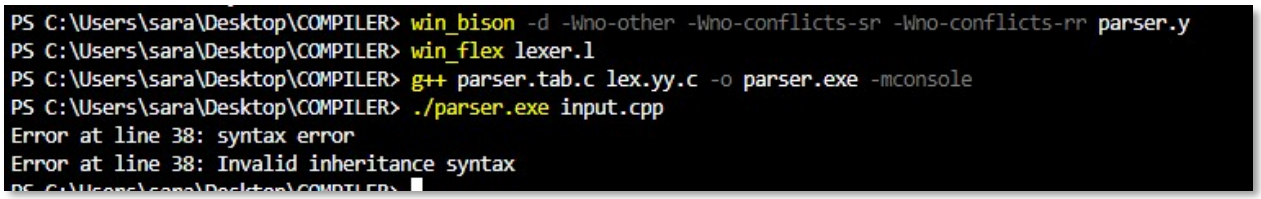


KEYWORD	20	cout
OPERATOR	20	<<
LITERAL_STRING	20	"Output statement"
OPERATOR	20	<<
KEYWORD	20	endl
PUNCTUATION	20	;
PUNCTUATION	21	}
KEYWORD	24	void
IDENTIFIER	24	inputName
PUNCTUATION	24	(
PUNCTUATION	24	)
PUNCTUATION	24	{
KEYWORD	25	cout
OPERATOR	25	<<
LITERAL_STRING	25	"Enter your name: "
PUNCTUATION	25	;
KEYWORD	26	cin
OPERATOR	26	>>
IDENTIFIER	26	name
PUNCTUATION	26	;
PUNCTUATION	27	}
KEYWORD	30	void
IDENTIFIER	30	greet
PUNCTUATION	30	(
PUNCTUATION	30	)
PUNCTUATION	30	{
KEYWORD	31	cout
OPERATOR	31	<<
LITERAL_STRING	31	"Hello, "
OPERATOR	31	<<
IDENTIFIER	31	name
OPERATOR	31	<<



Description: This shows tokens successfully parsed up until the inheritance issue disrupted the process.

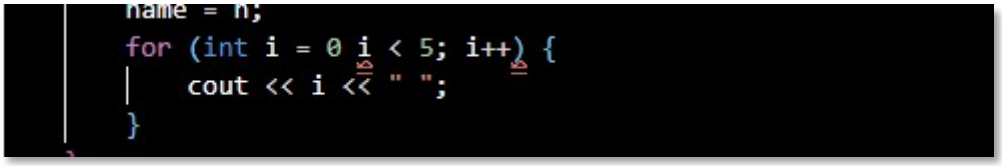
3. **Terminal Output (With Error):**
   Screenshot



   Description: The parser detects an error in the inheritance syntax and returns a corresponding error message.

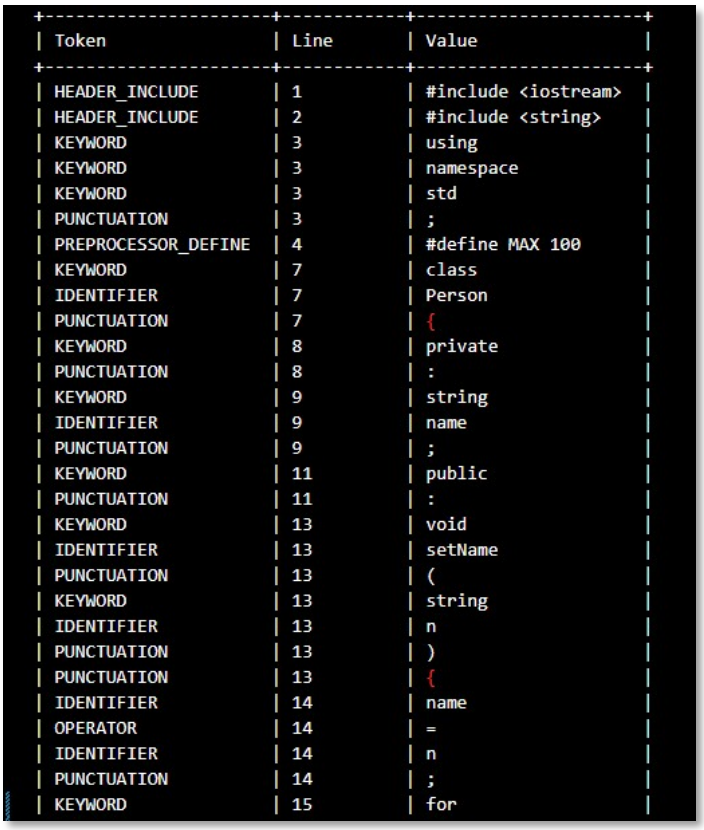
B.3: Loop Structure Error

1. **Input C++ File (With Error):**
   Screenshot

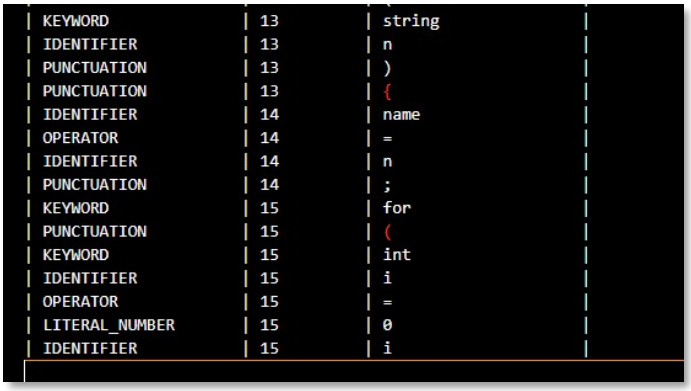


Description: This version contains a mistake in the loop declaration or body that prevents full parsing.

2. **Tokens Table (With Error):**
   Screenshot:



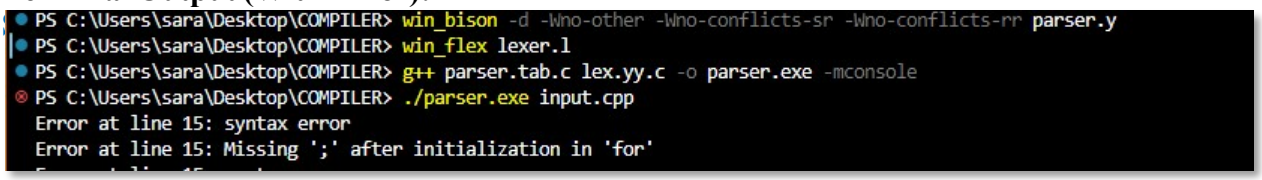
Token	Line	Value
HEADER_INCLUDE	1	#include <iostream>
HEADER_INCLUDE	2	#include <string>
KEYWORD	3	using
KEYWORD	3	namespace
KEYWORD	3	std
PUNCTUATION	3	;
PREPROCESSOR_DEFINE	4	#define MAX 100
KEYWORD	7	class
IDENTIFIER	7	Person
PUNCTUATION	7	{
KEYWORD	8	private
PUNCTUATION	8	:
KEYWORD	9	string
IDENTIFIER	9	name
PUNCTUATION	9	;
KEYWORD	11	public
PUNCTUATION	11	:
KEYWORD	13	void
IDENTIFIER	13	setName
PUNCTUATION	13	(
KEYWORD	13	string
IDENTIFIER	13	n
PUNCTUATION	13	)
PUNCTUATION	13	{
IDENTIFIER	14	name
OPERATOR	14	=
IDENTIFIER	14	n
PUNCTUATION	14	;
KEYWORD	15	for



KEYWORD	13	string
IDENTIFIER	13	n
PUNCTUATION	13	)
PUNCTUATION	13	{
IDENTIFIER	14	name
OPERATOR	14	=
IDENTIFIER	14	n
PUNCTUATION	14	;
KEYWORD	15	for
PUNCTUATION	15	(
KEYWORD	15	int
IDENTIFIER	15	i
OPERATOR	15	=
LITERAL_NUMBER	15	0
IDENTIFIER	15	i

Description: The tokens generated are listed up to the point where the loop structure breaks parsing.

3. **Terminal Output (With Error):**



```
PS C:\Users\sara\Desktop\COMPILER> win_bison -d -Wno-other -Wno-conflicts-sr -Wno-conflicts-rr parser.y
PS C:\Users\sara\Desktop\COMPILER> win_flex lexer.l
PS C:\Users\sara\Desktop\COMPILER> g++ parser.tab.c lex.yy.c -o parser.exe -mconsole
PS C:\Users\sara\Desktop\COMPILER> ./parser.exe input.cpp
Error at line 15: syntax error
Error at line 15: Missing ';' after initialization in 'for'
```

Description: The parser halts on encountering the loop error and outputs a message describing the issue.

B.4: Conditional Statement Error

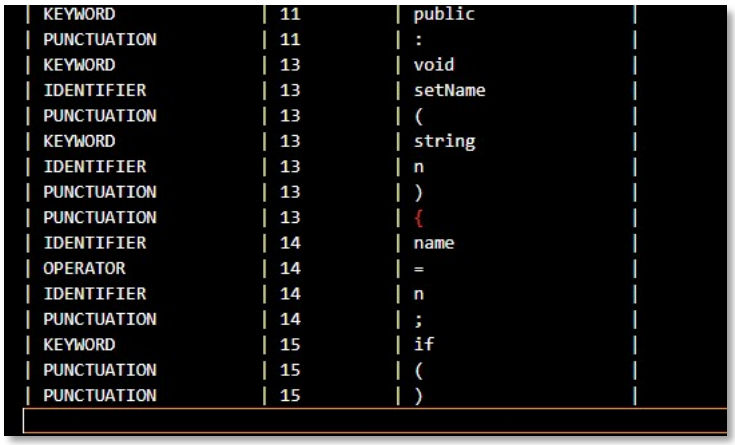
1. **Input C++ File (With Error):**
   Screenshot

```
    name = n;
    if () {  // Check if name is not empty
        name = n;
    } else {
        cout << "Error: Name cannot be empty" << endl;
    }
```

Description: The if condition in this code includes an issue that results in a syntax error during parsing.

2. **Tokens Table (With Error):**
   Screenshot:

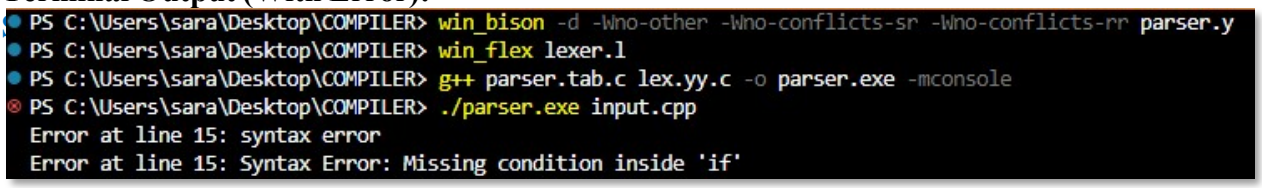
Token	Line	Value
HEADER_INCLUDE	1	#include <iostream>
HEADER_INCLUDE	2	#include <string>
KEYWORD	3	using
KEYWORD	3	namespace
KEYWORD	3	std
PUNCTUATION	3	;
PREPROCESSOR_DEFINE	4	#define MAX 100
KEYWORD	7	class
IDENTIFIER	7	Person
PUNCTUATION	7	{
KEYWORD	8	private
PUNCTUATION	8	:
KEYWORD	9	string
IDENTIFIER	9	name
PUNCTUATION	9	;
KEYWORD	11	public
PUNCTUATION	11	:
KEYWORD	13	void
IDENTIFIER	13	setName
PUNCTUATION	13	(
KEYWORD	13	string
IDENTIFIER	13	n
PUNCTUATION	13	)
PUNCTUATION	13	{
IDENTIFIER	14	name
OPERATOR	14	=
IDENTIFIER	14	n
PUNCTUATION	14	;
KEYWORD	15	if



KEYWORD	11	public
PUNCTUATION	11	:
KEYWORD	13	void
IDENTIFIER	13	setName
PUNCTUATION	13	(
KEYWORD	13	string
IDENTIFIER	13	n
PUNCTUATION	13	)
PUNCTUATION	13	{
IDENTIFIER	14	name
OPERATOR	14	=
IDENTIFIER	14	n
PUNCTUATION	14	;
KEYWORD	15	if
PUNCTUATION	15	(
PUNCTUATION	15	)

Description: Tokenization stops when the malformed conditional is encountered by the parser.

3. **Terminal Output (With Error):**

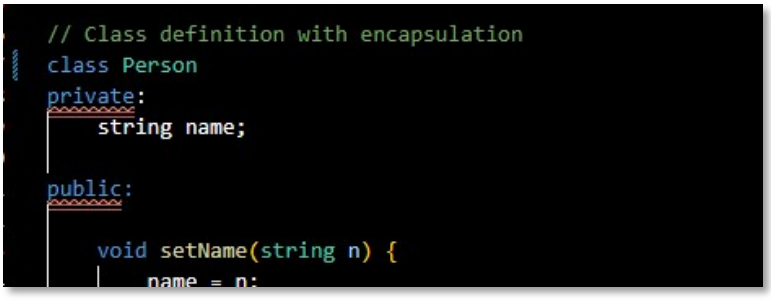


```
PS C:\Users\sara\Desktop\COMPILER> win_bison -d -Wno-other -Wno-conflicts-sr -Wno-conflicts-rr parser.y
PS C:\Users\sara\Desktop\COMPILER> win_flex lexer.l
PS C:\Users\sara\Desktop\COMPILER> g++ parser.tab.c lex.yy.c -o parser.exe -mconsole
PS C:\Users\sara\Desktop\COMPILER> ./parser.exe input.cpp
Error at line 15: syntax error
Error at line 15: Syntax Error: Missing condition inside 'if'
```

Description: An error message is displayed as the parser fails to interpret the conditional statement correctly.

B.5: Class Definition Error

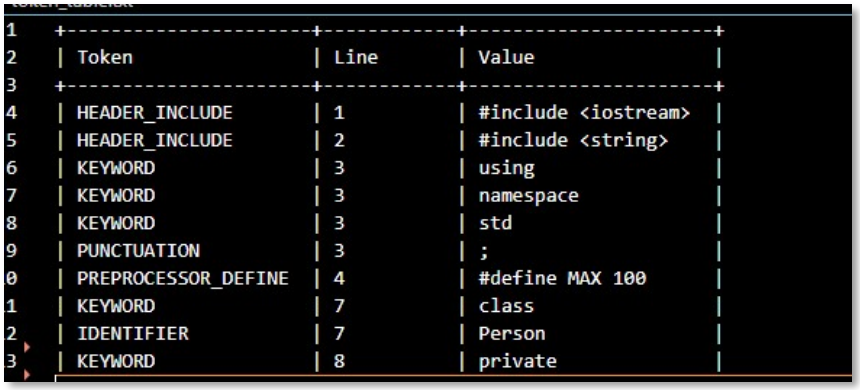
1. **Input C++ File (With Error):**
   Screenshot:



```
// Class definition with encapsulation
class Person
private:
    string name;

public:

    void setName(string n) {
        name = n;
```

Description: This version includes an error in the class structure that prevents proper parsing.

2. **Tokens Table (With Error):**
   Screenshot:

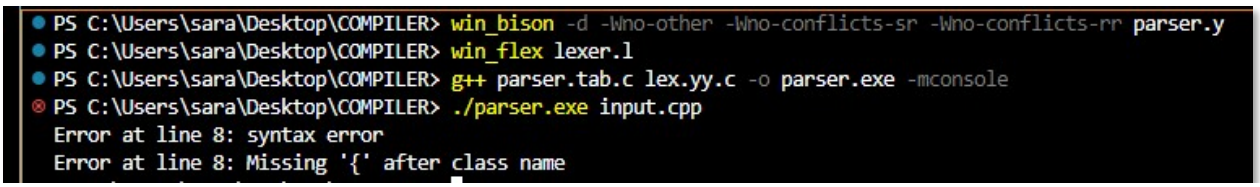


Token	Line	Value
HEADER_INCLUDE	1	#include <iostream>
HEADER_INCLUDE	2	#include <string>
KEYWORD	3	using
KEYWORD	3	namespace
KEYWORD	3	std
PUNCTUATION	3	;
PREPROCESSOR_DEFINE	4	#define MAX 100
KEYWORD	7	class
IDENTIFIER	7	Person
KEYWORD	8	private

Description: The token sequence ends prematurely due to the issue within the class definition.

3. **Terminal Output (With Error):**

Screenshot:



Description: The parser recognizes a class-related syntax error and generates an appropriate message.

10. Limitations:

- No support for custom types like struct, enum, and union unless explicitly defined.
- Type inference using auto is not recognized.
- Function overloading and template definitions are not supported.
- Reference variables (e.g., int& x = y;) are not supported.
- Features like decltype, typename, and using aliases are not handled.
- Only basic preprocessor directives are supported.
- Preprocessor matching is fragile and sensitive to whitespace and formatting.
- There is no implementation of scoping rules for variables or functions.
- Template constructs like template<typename T> are not recognized.
- Operator overloading (e.g., operator+, operator[]) is not supported.
- Standard Template Library (STL) types like vector, map, and set are not handled.
- Memory management keywords such as new, delete, malloc, and free are unsupported.
- Smart pointers like std::unique_ptr and std::shared_ptr are not supported.
- Lambda expressions (e.g., []() {}) are not handled.
- Many valid C++ keywords such as constexpr, noexcept, and mutable are not recognized.
- Operators such as ->, .*, new, delete, sizeof, and typeid are not supported.
- The design is rigid and relies on hardcoded patterns rather than modular rules.
- There is no configurability for different C++ standards like C++98, C++11, or C++20.

11. Assumptions:

- Source code can span multiple lines (not limited to a single line).
- All character and string literals are assumed to be correctly closed and well-formed.
- Preprocessor directives like #include, #define, #ifdef, etc., are expected to follow a strict and simple pattern.
- Whitespace and tabs are silently ignored and not tokenized.
- Newlines are only tracked for line number counting and not tokenized.
- Keywords like int, float, if, class, etc., are hardcoded and recognized only in lowercase.
- Identifiers must begin with a letter or underscore and may contain alphanumeric characters or underscores.
- Comments are completely ignored and not tokenized, with no support for malformed or unterminated comment detection.
- Each token must match exactly one pattern; no ambiguity resolution or backtracking is assumed.
- Escape sequences inside string and character literals are expected to be correct and valid.
- No distinction is made between user-defined types and keywords e.g., string is treated as a keyword.
- The lexer assumes input files are encoded in plain ASCII or UTF-8 without BOM or wide characters.