TOC GROUP Project

GROUP MEMBERS:

SYEDA AFIA NAEEM

YUSRA ABDUL JABBAR

MEHAK KAMRAN

MAHNOOR SARFARAZ

JUNAID UL ISLAM

- LANGUAGE USED:C++

1.VARIABLES:

LANGUAGE SPECIFICATION:

Variable/Identifier Rules:

1. Variable names must start with a letter or an underscore. They cannot start with a number.
2. Variable names can contain letters, numbers, and underscores.
3. Variable names are case-sensitive.
4. Variable names should be descriptive and meaningful.
5. Variable names should not be C++ keywords or reserved words.
6. Variable names should not include spaces or special characters.
7. Variable names should follow a consistent naming convention throughout your code.

REGULAR EXPRESSION:

A regular expression for a variable that allows lowercase and uppercase letters, digits, and underscores, and starts with a letter or underscore, can be written as:

^[a-zA-Z_][a-zA-Z0-9_]*$

- \`^\` - matches the start of the string
- \`[a-zA-Z_]\` - matches any lowercase letter, an uppercase letter, or underscore

- `[a-zA-Z0-9_]*` - matches zero or more occurrences of any lowercase letter, an uppercase letter, digit, or underscore

`$` - matches the end of the string

Therefore, this regular expression matches any string that starts with a letter or underscore, followed by zero or more occurrences of letters, digits, or underscores. This is a basic regular expression for a variable.

CFG:

<program> ::= <declaration>

<declaration> ::= <type> <identifier> ';'

<type> ::= 'int' | 'float' | 'double' | 'char' | 'bool'

<identifier> ::= <letter> <idchar>*

<letter> ::= 'a' | 'b' | ... | 'z' | 'A' | 'B' | ... | 'Z' | '_'

<idchar> ::= <letter> | <digit>

<digit> ::= '0' | '1' | ... | '9'

<assignment> ::= <variable> '=' <expression>

<expression> ::= <term> '+' <term>

<term> ::= <constant> | <variable>

<variable> ::= 'x' | 'y' | 'z'

<constant> ::= '0' | '1' | ... | '9'

In this CFG, a variable declaration consists of a type specifier (<type>), followed by an identifier (<identifier>), and ending with a semicolon (;). The type can be one of int, float, double, char, or bool. The identifier is a sequence of one or more letters, digits, or underscores, starting with a letter.

DERIVATION:

```
<program> => <declaration>

      => <type> <identifier> ';'

      => int <identifier> ';'

      => int myVar ';'
```

In this derivation, we start with the non-terminal symbol <program> and apply the production rule <program> => <declaration>. Then, we apply the production rule <declaration> ::= <type> <identifier> ';' to get <type> <identifier> ';'. We substitute int for <type> and myVar for <identifier> to get int myVar ';'. Finally, we substitute the terminal symbol ';' for ';' to get the complete variable declaration int myVar;. This derivation represents a valid way to generate a variable declaration in C++ using the grammar provided.

PARSE TREE:

```
    <program>
        |
  <declaration>
  /    |    \
<type> <identifier> ';'
 |      |     |
int    myVar   ;
```

In this parse tree, the <program> node represents the entire program, which consists of a single <declaration> node. The <declaration> node has three children: a <type> node, an <identifier> node, and a semicolon. The <type> node has a single child, which is the token "int", representing the integer data type. The <identifier> node has a single child, which is the token "myVar", representing the variable name. The semicolon is represented by a terminal symbol, which does not have any children. This parse tree represents the syntax of a valid C++ variable declaration, where the variable is named myVar and has type int.

ALORITHM:

To declare a variable you need to provide its name and data type. Here's an algorithm for declaring a variable

Choose a name for the variable, and ensure that it follows identifier naming rules.

Determine the data type for the variable, which can be one of the built-in types (such as int, char).

Write the declaration statement, which consists of the data type, followed by the variable name, and ends with a semicolon.

```
int myVar;
```

2.DATATYPES:

- **Numeric type:**
  1)Int(integer):
  LANGUAGE SPECIFICATION:
  In C++, an `int` is a data type that represents an integer number, which can be either positive, negative or zero.

  REGULAR EXPRESSION:
  Here is a regular expression of `int` in C++:

  [+-]?\d+

  Let's break down this regular expression:

  - [+-]? matches an optional plus or minus sign. The square brackets indicate a character set, which means that the plus and minus signs are treated as individual characters to match.
  - \d+ matches one or more digits. The backslash before `d` is an escape character that tells the regular expression engine to treat `d` as a digit, rather than a regular character. The plus sign means "one or more", so `\d+` matches any sequence of digits.

  Here are some examples of integers that match this regular expression:

-123
+9876543210
0
Note that this regular expression does not match integer values with commas, decimal points, or other non-digit characters. To match floating-point numbers or other numeric formats, you would need to modify the regular expression accordingly.

- Text type:
  2)Char(character):
  LANGUAGE SPECIFICATION:
  In C++, a `char` is a data type that represents a single character, enclosed in single quotes.

  REGULAR EXPRESSION:
  Here is a regular expression for `char` in C++:

  '[^\\']'

  Let's break down this regular expression:

  - The first and last single quotes match the beginning and end of the `char` literal.
  - `[^\\']` matches any character that is not a backslash or a single quote. The `^` character inside the square brackets negates the character set, so it matches any character except the ones listed.
  - The backslash before the single quote inside the square brackets is an escape character that tells the regular expression engine to treat the single quote as a regular character, rather than the end of the `char` literal.

  Here are some examples of `char` literals that match this regular expression:
  'a'
  'X'
  ' '
  '\n'
  '\\'
  Note that this regular expression assumes that the `char` literal does not contain any escape sequences or Unicode characters. To match more complex `char` literals, you would need to modify the regular expression accordingly.

3.<u>FOR LOOP:</u>

LANGUAGE SPECIFICATION:

The syntax of the for loop in C++ is as follows:

for (initialization ; condition ; increment) statement

The initialization expression is evaluated once at the beginning of the loop, before the first iteration. It typically initializes a loop counter or some other variable used in the loop.

The condition expression is evaluated at the beginning of each iteration. If the condition is true, the loop body is executed; otherwise, the loop terminates

The increment expression is evaluated at the end of each iteration, after the loop body has been executed. It typically updates the loop counter or some other variable used in the loop.

The statement is the code that is executed each iteration of the loop. It can be a single statement or a compound statement enclosed in curly braces.

Here is an example of a for loop in C++:

```
for (int i = 0; i < 10; i++) {
   std::cout << i << std::endl;}
```

This loop initializes the integer variable i to zero, checks if i is less than 10, and increments i by one after each iteration. The loop body consists of a single statement that prints the value of i to the console. The loop will execute 10 times, printing the numbers 0 through 9.

CFG:

Here is a possible CFG for the for loop syntax in C++:

```
<for-loop> ::= for ( <initialization> ; <condition> ; <increment> ) <statement>

<initialization> ::= <expression> | <declaration>

<condition> ::= <expression>

<increment> ::= <expression>

<statement> ::= <expression> | <compound-statement> | <empty-statement>

<compound-statement> ::= { <statement-seq> }

<statement-seq> ::= <statement> | <statement> <statement-seq>

<empty-statement> ::= ;
```

In this CFG, <expression> represents any valid C++ expression, and <declaration> represents any valid C++ variable declaration.

The <initialization> nonterminal can be either an expression or a declaration. In the case of an expression, it is evaluated once at the beginning of the loop. In the case of a declaration, the variable is declared and initialized once at the beginning of the loop.

The <condition> nonterminal represents the loop condition, which is evaluated before each iteration of the loop.

The <increment> nonterminal represents the loop increment expression, which is evaluated at the end of each iteration of the loop.

The <statement> nonterminal represents the code that is executed during each iteration of the loop. It can be a single expression, a compound statement (enclosed in curly braces), or an empty statement (represented by a semicolon).

Parsing:

Here is an example of a parse tree for a simple for loop in C++:

```
for ( int i = 0 ; i < 10 ; i++ ) {

   std::cout << i << std::endl;

}
```

```
<for-loop>

          / | \

  "for"    /  |  \   <statement>

         /   |   \   |

    "("  /    |    \  |

      / <initialization> \

     /     /  |  \      \

 "int"  "i"   "=" "0"  ";" <expression>  <increment>

                         |

                     <expression>

                         |

                      "i++"
                     |
                    "i"
```

This parse tree shows the structure of the for loop in terms of the non terminals defined in the CFG for the for loop.

The root of the tree is labeled <for-loop>, which represents the entire for loop.

The children of the root node are labeled <initialization>, <condition>, <increment>, and <statement>, corresponding to the four parts of the for loop syntax.

The <initialization> node has three children, representing the variable declaration, the variable name, and the initialization expression.

The <condition> node has one child, representing the boolean expression that determines when the loop should terminate.

The <increment> node has one child, representing the expression that is evaluated at the end of each iteration of the loop.

The <statement> node has one child, representing the code that is executed during each iteration of the loop. In this case, it is a single expression that prints the value of i to the console.

The parse tree can be used to generate a derivation of the input string according to the rules of the CFG. Each node in the tree corresponds to a nonterminal in the CFG, and each edge corresponds to a production rule in the CFG.

Derivation:

Here is an example derivation of a simple for loop in C++ using the CFG above

<for-loop> -> for (<initialization> ; <condition> ; <increment>) <statement>

-> for (<expression> ; <expression> ; <expression>) <statement>

-> for (<declaration> ; <expression> ; <expression>) <statement>

-> for ("int" <identifier> "=" <integer-literal> ";" <expression> ";" <expression>) <statement>

-> for ("int" "i" "=" "0" ";" <expression> ";" <expression>) <statement>

-> for ("int" "i" "=" "0" ";" <identifier> "<" <integer-literal> ";" <expression>) <statement>

-> for ("int" "i" "=" "0" ";" "i" "<" "10" ";" <expression>) <statement>

-> for ("int" "i" "=" "0" ";" "i" "<" "10" ";" "i++") <statement>

-> for (<expression> ; <expression> ; <expression>) <compound-statement>

-> for (<identifier> "<" <integer-literal> ";" <expression>) <compound-statement>

-> for ("i" "<" "10" ";" "i++") <compound-statement>

-> for (<expression>) <compound-statement>

-> for (<expression>) { <statement-seq> }

-> for (<expression>) { <statement> }

-> for (<expression>) { <expression> }

-> for (<expression>) { <std::cout> << <expression> << <std::endl> ; }

This derivation shows the step-by-step process of rewriting the initial input string using the production rules of the CFG. At each step, a nonterminal is replaced by one of its production rules, until only terminals (such as keywords, identifiers, and literals) remain.

The final result is a parse tree that represents the structure of the input string in terms of the nonterminals defined in the CFG.

Algorithm:

Here is the algorithm of a for loop in C++:

Evaluate the expression in the initialization clause, if present. If not present, go to step 4.

If the result of the evaluation is an lvalue, apply lvalue-to-rvalue conversion to obtain the value of the object referred to.

Create a new scope with the object(s) declared in the initialization clause.

Evaluate the expression in the condition clause. If it is true, execute the statement(s) in the body of the loop.

Evaluate the expression in the increment clause, if present.

Go to step 4.

The steps can be summarized as follows:

Initialization

Evaluate the expression in the initialization clause, if present.

If the result is an lvalue, apply lvalue-to-rvalue conversion.

Create a new scope with the object(s) declared in the initialization clause.

Condition

Evaluate the expression in the condition clause.

If it is true, execute the statement(s) in the body of the loop.

Increment

Evaluate the expression in the increment clause, if present.

Repeat from step 2.

Note that the initialization clause, condition clause, and increment clause can be any expressions, not just simple variable assignments or comparisons. For example, they can include function calls, nested loops, or complex conditionals.

4.<u>OPERATORS:</u>

Language Specification:

Here are the language specifications of the operators used in the example for loop in C++:
= (assignment operator): assigns the value of the right operand to the left operand.
+ (addition operator): performs addition on two operands.
< (less than operator): returns true if the left operand is less than the right operand, otherwise false.
++ (increment operator): increments the value of the operand by 1.
<< (output operator): outputs the value of the right operand to the left operand (typically a stream object).
std::cout and std::endl are not operators, but rather part of the standard library in C++ that provide input/output functionality.
It is important to note that the semantics and precedence of these operators may vary depending on their context and the types of operands involved. For example, the + operator can also perform string concatenation when applied to std::string objects, and the << operator can be overloaded for custom output formatting.

CFG:

```
<for-loop> ::= for (<initialization> ; <condition> ; <increment/decrement>) <loop-body>
<initialization> ::= <data-type> <identifier> = <expression>
<condition> ::= <expression>
<increment/decrement> ::= <identifier> ++ | <identifier> --
<loop-body> ::= { <statements> }

<data-type> ::= int | char | float | double | long | short | unsigned
<identifier> ::= <letter> | <identifier> <letter> | <identifier> <digit>
<letter> ::= a | b | c | ... | z | A | B | C | ... | Z | _
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<expression> ::= <identifier> <arithmetic-operator> <identifier> | <identifier>
<comparison-operator> <identifier> | <constant>
<arithmetic-operator> ::= + | - | * | / | %
<comparison-operator> ::= == | != | < | > | <= | >=
<constant> ::= <integer-constant> | <floating-point-constant> | <character-constant> |
<string-constant>

<integer-constant> ::= <digit> | <integer-constant> <digit>
<floating-point-constant> ::= <integer-constant> . <integer-constant>
<character-constant> ::= '<letter>' | '<digit>' | '<special-character>'
<string-constant> ::= "<string-characters>"
```

<string-characters> ::= <letter> | <digit> | <special-character> | <string-characters> <letter> | <string-characters> <digit> | <string-characters> <special-character>
<special-character> ::= ! | @ | # | $ | % | ^ | & | * | (|) | _ | + | - | = | { | } | [|] | ; | : | ' | " | < | > | , | . | / | \ | |

Parsing:
how the operators in the for loop example are parsed:
1. The for loop starts with the "for" keyword, which is followed by a set of parentheses.
2. Inside the parentheses, the initialization statement "int i = 0" is used to declare and initialize a variable called "i" of type "int" with an initial value of 0.
3. Next, the loop condition "i < 10" is used to specify the condition that needs to be true for the loop to continue iterating.
4. After the loop condition, there is a semicolon that separates the loop condition from the third part of the for loop.
5. The third part of the for loop, "i++", is used to increment the value of the variable "i" after each iteration of the loop.
6. The code block enclosed in curly braces "{" and "}" contains the statements that will be executed repeatedly during each iteration of the loop.

Parse Tree:

```
for
        |
    +----+----+
    |        |
  int        |
    |        |
    i        |
    |        |
    =        0
    |        |
+----+----+    |
|        |    |
<       10    |
|        |    |
|        |    |
i       ++    |
|            |
|            |
{            }
```

|
|
code block
This parse tree shows the hierarchical relationships between the operators in the for loop example. The top level is the "for" keyword, which has three child nodes representing the initialization statement, loop condition, and increment statement. The loop condition has two child nodes, which are the less than operator and the constant value 10. The increment statement has one child node, which is the increment operator "++". The parse tree also shows that the code block is a child node of the for loop, indicating that the statements inside the code block are executed during each iteration of the loop.

Derivation:
for_loop -> for (initialization ; loop_condition ; increment) statement

initialization -> declaration ;

declaration -> type_specifier identifier

type_specifier -> int

identifier -> i
initialization -> int i ;
loop_condition -> expression
expression -> binary_expression
binary_expression -> identifier < constant
identifier -> i
constant -> 10
loop_condition -> i < 10
increment -> expression_statement
expression_statement -> postfix_expression ++ ;
postfix_expression -> identifier
identifier -> i
increment -> i++ ;
statement -> compound_statement
compound_statement -> { statement_list }
statement_list -> /* code to be executed */
for_loop -> for (int i = 0 ; i < 10 ; i++) { /* code to be executed */ }

Algorithm:
1. Initialization: The initialization expression is executed once before the loop begins. This expression is used to initialize the loop variable, which controls the number of iterations. For example: for (int i = 0; i < 10; i++), initializes the loop variable i to 0.
2. Condition: The condition is evaluated at the beginning of each iteration. The loop will continue to execute as long as the condition is true. If the condition is false, the loop will exit. For example: i < 10 is the condition in the previous example.
3. Increment/Decrement: The increment/decrement expression is executed at the end of each iteration. This expression is used to update the loop variable. For example: i++ increments the value of i by 1 at the end of each iteration.
4. Loop Body: The loop body contains the statements that are executed repeatedly until the loop terminates. For example, if you want to print the value of i in each iteration, you would put the statement cout << i << endl; inside the loop body.
5. Exit Loop: Once the condition evaluates to false, the loop terminates and execution continues with the statement following the loop.

Keywords:

LANGUAGE SPECIFICATION:

and	and_eq	asm	auto	bitand	bitor	break
if	for	int	static	else	not	export
union	true	char	goto	enum	return	volatile
void	volatile	continue	unsigned	false	static	try
catch	class	compl	const	constexpr	const_cast	decltype
default	delete	dynamic_cast	explicit	export	extern	friend
inline	mutable	namespace	new	noexcept	not_eq	nullptr

or	private	protected	public	register	sizeof	register

Algorithm:

Here's an algorithm to recognize a keyword :

1. Identify the word to be checked.
2. Compare the word to the list of keywords in the language specification.
3. If the word matches a keyword, then it is a keyword; otherwise, it is not.

5.<u>CONDITIONAL STATEMENT:</u>

Language Specification:

- **if-else statement**

syntax:

```
if (condition) {
   // code to execute if condition is true
} else {
   // code to execute if condition is false
}
```

The if-else statement in C++ is used to execute one block of code if a condition is true and another block of code if the condition is false. The syntax is "if (condition) { /* code if true / } else { / code if false */ }". The "else" block is optional. The "condition" is any expression that can be evaluated as true or false. If the condition is true, the code inside the "if" block is executed; otherwise, the code inside the "else" block (if present) is executed.

Example of an if-else statement in C++:

```
#include <iostream>


int main() {
    int num = 15;


    if (num > 10) {
        std::cout << "The number is greater than 10." << std::endl;
    } else {
        std::cout << "The number is less than or equal to 10." << std::endl;
    }


    return 0;
}
```

CFG:

```
S -> if (E) S else S | other_statements
E -> ID > NUM
```

Here, S represents a statement and E represents an expression. The CFG states that a statement S can be an if-else statement or any other statements. An expression E can be a comparison of an identifier (ID) with a number (NUM) using the greater than (>) operator.

Parsing :

The parsing process is about breaking down the code into a hierarchical structure that follows the CFG.

Token stream: if (ID > NUM) { other_statements } else { other_statements }

```
S -> if ( E ) S else S
  -> if ( ID > NUM ) S else S
  -> if ( num > NUM ) S else S
  -> if ( num > 10 ) S else S
```

-> if (num > 10) { other_statements } else { other_statements }

Here, we have used the CFG to parse the code by applying the productions rules to the token stream. We start with the start symbol S and derive the input code to a set of non-terminals and terminals that follow the CFG.

Parse tree:

```
    S
  ____|__________________________
 |                    |
 if                  else
 |                    |
 (E)                  S
 |                    |
 ID > NUM             ___________|____________
                 |               |
                {S}             {S}
                 |               |
          other_statements    other_statements
```

In this parse tree, S represents the top-level statement, if, else, and other_statements are terminals, and E represents an expression. The expression is represented by the subtree rooted at (E), which has two children representing the ID and NUM tokens. The if statement has a left child of (E) and a right child representing the statement inside the if block, which is itself a subtree rooted at {S}. The else statement has a left child of {S} and a right child representing the statement inside the else block, which is also a subtree rooted at {S}.

Derivation:

Suppose we have the following if-else statement in C++ using the CFG provided:

```
if (x > 5) {
   y = 10;
} else {
   y = 5;
}
```

To derive this statement using the CFG S -> if (E) S else S | other_statements and E -> ID > NUM, we start with the start symbol S and apply the production rules to generate the statement:

```
S      -> if (E) S else S
       -> if (ID > NUM) S else S
       -> if (x > NUM) S else S
       -> if (x > 5) S else S
       -> if (x > 5) { other_statements } else { other_statements }
       -> if (x > 5) { y = 10; } else { other_statements }
       -> if (x > 5) { y = 10; } else { y = 5; }
```

Here, we have used the CFG rules to derive the input code to a parse tree. The derivation shows the exact steps that were taken to generate the code from the start symbol S.

Algorithm:

1) Declare an integer variable num and initialize it to the value of 15.

2) Evaluate the expression num > 10.

3) If the expression is true, output the message "The number is greater than 10." to the console using the std::cout statement. Otherwise, output the message "The number is less than or equal to 10." to the console.

4) End the if-else statement.

5) Return the value of 0 to indicate successful program execution.

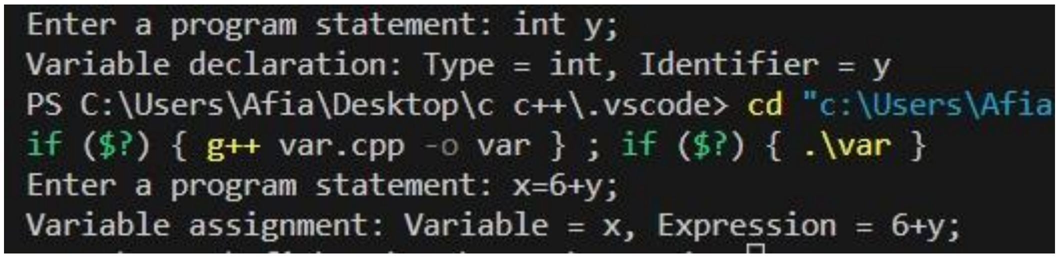
Implementation

Variable declaration and initialization

Code:

```
#include <iostream>
#include <string>

bool isValidIdentifier(const std::string& str) {
    if (str.empty() || !isalpha(str[0]) && str[0] != '_')
        return false;

    for (size_t i = 1; i < str.length(); i++) {
        if (!isalnum(str[i]) && str[i] != '_')
            return false;
    }

    return true;
}

int main() {
    std::string input;

    // Prompt for input
    std::cout << "Enter a program statement: ";
    std::getline(std::cin, input);

    // Parsing
    size_t pos = 0;

    // Remove leading whitespaces
    while (pos < input.length() && isspace(input[pos]))
        pos++;

    // Check if it's a declaration
    if (input.substr(pos, 3) == "int" || input.substr(pos, 5) == "float" ||
input.substr(pos, 6) == "double" ||
        input.substr(pos, 4) == "char" || input.substr(pos, 4) == "bool") {

        std::string type;

        // Extract the type
        while (pos < input.length() && !isspace(input[pos])) {
            type += input[pos];
            pos++;
        }

        // Skip whitespaces after the type
        while (pos < input.length() && isspace(input[pos]))
            pos++;

        std::string identifier;

        // Extract the identifier
        while (pos < input.length() && (isalnum(input[pos]) || input[pos] ==
'_')) {
            identifier += input[pos];
            pos++;
        }

        // Check if the identifier is valid
        if (!isValidIdentifier(identifier)) {
            std::cout << "Invalid identifier!" << std::endl;
            return 0;
        }

        // Skip whitespaces after the identifier
        while (pos < input.length() && isspace(input[pos]))
            pos++;

        // Check for the semicolon at the end of the declaration
        if (input[pos] == ';') {
            std::cout << "Variable declaration: Type = " << type << ", Identifier
= " << identifier << std::endl;
        } else {
            std::cout << "Missing semicolon!" << std::endl;
        }
    }
    // Check if it's an assignment
    else {
        std::string variable;
        std::string expression;

        // Extract the variable
        if (input[pos] == 'x' || input[pos] == 'y' || input[pos] == 'z') {
            variable += input[pos];
            pos++;
        } else {
            std::cout << "Invalid variable!" << std::endl;
            return 0;
        }

        // Skip whitespaces after the variable
        while (pos < input.length() && isspace(input[pos]))
            pos++;

        // Check for the equals sign
        if (input[pos] != '=') {
            std::cout << "Missing equals sign!" << std::endl;
            return 0;
        }

        pos++; // Skip the equals sign

        // Skip whitespaces after the equals sign
        while (pos < input.length() && isspace(input[pos]))
            pos++;

        // Extract the expression
        while (pos < input.length()) {
            expression += input[pos];
            pos++;
        }

        std::cout << "Variable assignment: Variable = " << variable << ", 
Expression = " << expression << std::endl;
    }

    return 0;
}
```

Output:



```
Enter a program statement: int y;
Variable declaration: Type = int, Identifier = y
PS C:\Users\Afia\Desktop\c c++\.vscode> cd "c:\Users\Afia
if ($?) { g++ var.cpp -o var } ; if ($?) { .\var }
Enter a program statement: x=6+y;
Variable assignment: Variable = x, Expression = 6+y;
```

For Loop Implementation:

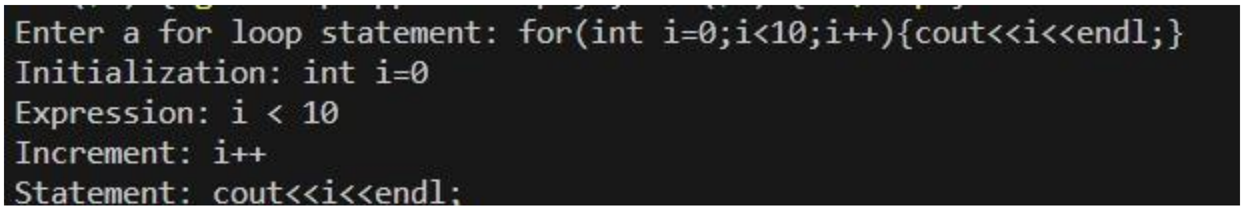
Code:

```cpp
#include <iostream>
#include <string>
#include <cctype>

struct Assignment {
    std::string statement;
};

struct Expression {
    std::string identifier;
    int number;
};

bool parseAssignment(const std::string& str, Assignment& assignment) {
    assignment.statement = str;
    return true; // No validation is performed in this example
}

bool parseExpression(const std::string& str, Expression& expression) {
    std::string::const_iterator it = str.begin();
    std::string identifier;

    // Parse the identifier
    while (it != str.end() && std::isalpha(*it)) {
        identifier.push_back(*it);
        ++it;
    }
    if (identifier.empty()) {
        return false;
    }
    expression.identifier = identifier;

    // Check for the less than sign
    if (it == str.end() || *it != '<') {
        return false;
    }
    ++it;

    // Parse the number
    int number = 0;
    while (it != str.end() && std::isdigit(*it)) {
        number = number * 10 + (*it - '0');
        ++it;
    }
    if (it != str.end()) {
        return false;
    }
    expression.number = number;

    return true;
}




int main() {
    std::string input;

    // Prompt for input
    std::cout << "Enter a for loop statement: ";
    std::getline(std::cin, input);

    // Parsing
    std::string::size_type pos = input.find("for");
    if (pos == std::string::npos) {
        std::cout << "Invalid for loop statement!" << std::endl;
        return 0;
    }
    pos += 3; // Skip "for"

    // Skip whitespaces
    while (pos < input.length() && std::isspace(input[pos])) {
        ++pos;
    }

    // Check for the opening parenthesis
    if (pos == input.length() || input[pos] != '(') {
        std::cout << "Invalid for loop statement!" << std::endl;
        return 0;
    }
    ++pos;

    // Skip whitespaces
    while (pos < input.length() && std::isspace(input[pos])) {
        ++pos;
    }

    // Parse the assignment
    Assignment assignment;
    std::string assignmentStr;
    while (pos < input.length() && input[pos] != ';') {
        assignmentStr.push_back(input[pos]);
        ++pos;
    }
    if (!parseAssignment(assignmentStr, assignment)) {
        std::cout << "Invalid assignment!" << std::endl;
        return 0;
    }

    // Skip the semicolon
    ++pos;

    // Skip whitespaces
    while (pos < input.length() && std::isspace(input[pos])) {
        ++pos;
    }

    // Parse the expression
    Expression expression;
    std::string expressionStr;
    while (pos < input.length() && input[pos] != ';') {
        expressionStr.push_back(input[pos]);
        ++pos;
    }
    if (!parseExpression(expressionStr, expression)) {
        std::cout << "Invalid expression!" << std::endl;
        return 0;
    }

    // Skip the semicolon
    ++pos;

    // Skip whitespaces
    while (pos < input.length() && std::isspace(input[pos])) {
        ++pos;
    }

    // Parse the increment
    Assignment increment;
    std::string incrementStr;
```

```cpp
    while (pos < input.length() && input[pos] != ')') {
        incrementStr.push_back(input[pos]);
        ++pos;
    }
    if (!parseAssignment(incrementStr, increment)) {
        std::cout << "Invalid increment!" << std::endl;
        return 0;
    }

    // Skip the closing parenthesis
    ++pos;

    // Skip whitespaces
    while (pos < input.length() && std::isspace(input[pos])) {
        ++pos;
    }

    // Check for theopening brace
if (pos == input.length() || input[pos] != '{') {
std::cout << "Invalid for loop statement!" << std::endl;
return 0;
}
++pos;
// Skip whitespaces
while (pos < input.length() && std::isspace(input[pos])) {
    ++pos;
}

// Parse the statement
std::string statementStr;
while (pos < input.length() && input[pos] != '}') {
    statementStr.push_back(input[pos]);
    ++pos;
}

// Check for the closing brace
if (pos == input.length() || input[pos] != '}') {
    std::cout << "Invalid for loop statement!" << std::endl;
    return 0;
}

// Output the parsed elements
std::cout << "Initialization: " << assignment.statement << std::endl;
std::cout << "Expression: " << expression.identifier << " < " <<
expression.number << std::endl;
```

```cpp
std::cout << "Increment: " << increment.statement << std::endl;
std::cout << "Statement: " << statementStr << std::endl;

return 0;
}
```

Output:



```
Enter a for loop statement: for(int i=0;i<10;i++){cout<<i<<endl;}
Initialization: int i=0
Expression: i < 10
Increment: i++
Statement: cout<<i<<endl;
```