Ahsanullah University of Science and Technology (AUST)
Department of Computer Science and Engineering

Assignment 5

Course No.: CSE4130

Course Title: Formal Languages & Compilers Lab

Date of Submission-09.08.2023

Submitted To- Mr. Md. Aminur Rahman & Iffatur Nessa.

Submitted By-

MD Shihabul Islam Shovo
190204075
Group: B1
Year- 4th
Semester- 1st
Session: Fall'22
Department- CSE

Answer:

```
#include <iostream>
#include <string>
#include <regex>
using namespace std;
string input_str;
string input_expr;
int idx = 0;
int position = 0;
int counter = 0;
regex identifier("^[a-zA-Z_][a-zA-Z0-9_]*$");
regex num_literal(R"(-?\d+(\.\d+)?([eE][+-]?\d+)?)");
int evaluate_statement();
int evaluate_expression();
int evaluate_term();
bool is_valid_identifier(const string& str)
{
   static regex identifier("^[a-zA-Z_][a-zA-Z0-9_]*$");
   return regex_match(str, identifier);
}
//change
bool is_valid_num_literal(const string& str)
{
   return regex_match(str, num_literal);
}
int evaluate_expression();
int evaluate_factor()
{
```

```
    string x;
    switch (input_str[idx])
    {
        case '(':
            idx++; // Move to the next character ('(' was found)
            if (evaluate_expression() && input_str[idx] == ')')
            {
                idx++; // Move to the next character (')' was found)
                return 1;
            }
            return 0;
        default:
            while (isalnum(input_str[idx]))
            {
                x.push_back(input_str[idx]);
                idx++;
            }
            if (is_valid_identifier(x) || is_valid_num_literal(x))
            {
                return 1;
            }
            return 0;}}
int evaluate_expression()
{
    int f = evaluate_factor();

    while (idx < input_str.length() && f == 1)
    {
```

```
        char op = input_str[idx];
        if (op == '*' || op == '/')
        {
            idx++; // Move to the next character (operator was found)
            int nextFactor = evaluate_factor();
            f = (nextFactor == 1) ? 1 : 0;
        }
        else if (op == '+' || op == '-')
        {
            idx++; // Move to the next character (operator was found)
            int nextTerm = evaluate_factor();
            f = (nextTerm == 1) ? 1 : 0;
        }
        else
        {
            break; // No more valid operators, exit the loop
        }}
    return f;
}
int evaluate_simple_expression()
{
    int f = 0;
    f = evaluate_expression();
    return f;
}
int evaluate_relop()
{
    if (input_str[idx] == '=' && input_str[idx + 1] == '=')
```

```
    {
        idx += 2; // Move to the next character (operator was found)
        return 1;
    }
    else if (input_str[idx] == '!' && input_str[idx + 1] == '=')
    {
        idx += 2; // Move to the next character (operator was found)
        return 1;
    }
    else if (input_str[idx] == '>' && input_str[idx + 1] == '=')
    {
        idx += 2; // Move to the next character (operator was found)
        return 1;
    }
    else if (input_str[idx] == '<' && input_str[idx + 1] == '=')
    {
        idx += 2; // Move to the next character (operator was found)
        return 1;
    }
    else if (input_str[idx] == '>' || input_str[idx] == '<')
    {
        idx++; // Move to the next character (operator was found)
        return 1;
    }else
    {return 0;}}
int evaluate_extension()
{
    if (idx >= input_str.length())
```

```
    {
        return 1; // Expression ends here, return 1 to indicate success
    }
    int f = evaluate_relop();
    if (f == 1)
    {
        return evaluate_simple_expression() ? 1 : 0;
    }
    return 1; // No comparison operator found, return 1 to indicate success
}
int evaluate_expression_extension()
{
    int f = 0;
    f = evaluate_simple_expression();
    if (f == 1)
    {f = evaluate_extension();}
    return f;}
int evaluate_assignment_statement()
{
   string x;
    // Parse the identifier
    while (isalnum(input_str[idx]))
    {
        x.push_back(input_str[idx]);
        idx++;
    }
    // Check if the identifier is valid
    if (!regex_match(x, identifier))
```

```
    {
        return 0; // Invalid identifier, return 0 to indicate failure
    }
    // Check for the assignment operator '='
    if (input_str[idx] == '=')
    {
        idx++; // Move to the next character (operator '=' was found)
    }
    else
    {
        return 0; // Missing assignment operator, return 0 to indicate failure
    }
    // Evaluate the expression on the right side of the assignment
    int f = evaluate_expression_extension();
    return f;
}
int evaluate_extension_1()
{
    if (idx >= input_str.length())
    {
        return 1; // Expression ends here, return 1 to indicate success
    }

    int z = idx;
    string x;

    // Parse the next word
    while (isalnum(input_str[idx]))
```

```
    {
        x.push_back(input_str[idx]);
        idx++;
    }

    if (x == "else")
    {
        idx++; // Move to the next character ('else' was found)

        // Evaluate the statement after 'else'
        if (evaluate_statement())
        {
            return 1; // The 'else' statement is valid, return 1 to indicate success
        }}
    idx = z; // Reset the index to the original position
    return 1; // No 'else' statement found, return 1 to indicate success
}
int evaluate_decision_statement()
{
    string x;

    // Parse the next word
    while (isalnum(input_str[idx]))
    {
        x.push_back(input_str[idx]);
        idx++;
    }
```

```
   if (x == "if")
   {
      cout<<"hdh";
      // Check for '(' after 'if'
      if (input_str[idx++] == '(')
      {
         // Evaluate the expression inside the parentheses
         if (evaluate_expression_extension())
         {
            // Check for ')' after the expression
            if (input_str[idx++] == ')')
            {
               // Evaluate the statement after the if condition
               if (evaluate_statement())
               {
                  // Evaluate the extension_1 (optional 'else' part)
                  if (evaluate_extension_1())
                  {
                     return 1; // The 'if' statement is valid, return 1 to indicate success
}}}}}}
   return 0; // The 'if' statement is invalid or not found, return 0 to indicate failure
}
int evaluate_loop_statement()
{
   string x;
   // Parse the next word
   while (isalnum(input_str[idx]))
   {
```

```
        x.push_back(input_str[idx]);
        idx++;
    }
    if (x == "while")
    {
        // Check for '(' after 'while'
        if (input_str[idx++] == '(')
        {
            // Evaluate the expression inside the parentheses
            if (evaluate_expression_extension() && input_str[idx++] == ')')
            {
                // Evaluate the statement inside the loop
                if (evaluate_statement())
                {
                    return 1; // The 'while' loop is valid, return 1 to indicate success
}}}}
    else if (x == "for")
    {
        // Check for '(' after 'for'
        if (input_str[idx++] == '(')
        {
            // Evaluate the initialization statement for the loop
            if (evaluate_assignment_statement() && input_str[idx++] == ';')
            {
                // Evaluate the expression for the loop condition
                if (evaluate_expression_extension() && input_str[idx++] == ';')
                {
                    // Evaluate the update statement for the loop
```

```
            if (evaluate_assignment_statement() && input_str[idx++] == ')')
            {
               // Evaluate the statement inside the loop
               if (evaluate_statement())
               {
                  return 1; // The 'for' loop is valid, return 1 to indicate success
               }}}}}}
   return 0; // The loop statement is invalid or not found, return 0 to indicate failure
}
int evaluate_statement()
{
   string x1;
   int id1 = 0;
   // Parse the next word
   while (isalnum(input_str[id1]))
   {
      x1.push_back(input_str[id1]);
      id1++;}
   int y ;
   position = idx;
   if (evaluate_assignment_statement())
   {
      idx++;
      return 1; // Assignment statement is valid, return 1 to indicate success
   }
   idx = position; // Reset the index to the original position
  if(x1=="while"|| x1=="for") y = evaluate_loop_statement();
  if(x1=="if") y = evaluate_decision_statement();
```

```
    if(y)
    {
        return 1; // Decision or loop statement is valid, return 1 to indicate success
    }
    return 0; // Statement is invalid, return 0 to indicate failure
}
int main()
{
    string s = "ad";
    regex pattern("^a(b(b|c))*d$");
    if (regex_match(s, pattern)) {
        cout << "String accepted" << std::endl;
    } else {
        cout << "String not accepted" << std::endl;
    }
    input_str = "b=b*c+b*c for(a=b; a<n; b=c) if(a<n) b= b*c+b*c";
    cout << "Statement: " << input_str << endl;
    if (evaluate_statement())
        cout << "Statement is correct." << endl;
    else
        cout << "Statement is incorrect." << endl;

    return 0;
}
```