Compiladores\lab4\lab4_plantilla.cpp

```cpp
#include <iostream>
#include <cstring>
#include <string>
#include <fstream>
#include <vector>
#include <iomanip>

using namespace std;

class Token {
public:
    enum Type { LPAREN=0, RPAREN, PLUS, MINUS, MULT, DIV, POW, NUM, ERR, END, ID, SIN, COS, LOG, PI, E, SEMICOLON };
    static const char* token_names[17];
    Type type;
    string lexema;
    int line;
    Token(Type);
    Token(Type, char c);
    Token(Type, const string source);
    Token(Type, int line);
    Token(Type, char c, int line);
    Token(Type, const string source, int line);
};

const char* Token::token_names[17] = { "LPAREN", "RPAREN", "PLUS", "MINUS", "MULT", "DIV", "POW", "NUM", "ERR", "END", "ID", "SIN", "COS", "LOG", "PI", "E", "SEMICOLON" };

Token::Token(Type type):type(type) { lexema = ""; line = 0; }
Token::Token(Type type, char c):type(type) { lexema = c; line = 0; }
Token::Token(Type type, const string source):type(type) { lexema = source; line = 0; }
Token::Token(Type type, int line):type(type), line(line) { lexema = ""; }
Token::Token(Type type, char c, int line):type(type), line(line) { lexema = c; }
Token::Token(Type type, const string source, int line):type(type), line(line) { lexema = source; }

std::ostream& operator << ( std::ostream& outs, const Token & tok ) {
    if (tok.lexema.empty())
        return outs << Token::token_names[tok.type];
    else
        return outs << Token::token_names[tok.type] << "(" << tok.lexema << ")";
}

std::ostream& operator << ( std::ostream& outs, const Token* tok ) {
    return outs << *tok;
}

class Scanner {
public:
    Scanner(const char* in_s);
    Token* nextToken();
    Token* nextTokenWithLine();
```

```
    ~Scanner();
    void printTokenTable();
private:
    string input;
    int first, current;
    int currentLine;
    vector<Token*> tokensTable;
    char nextChar();
    void rollBack();
    void startLexema();
    string getLexema();
    bool isNumberChar(char c);
};

Scanner::Scanner(const char* s):input(s),first(0),current(0),currentLine(1) { }

bool Scanner::isNumberChar(char c) {
    return isdigit(c) || c == '.';
}

Token* Scanner::nextToken() {
    char c;
    startLexema();

    while (true) {
        c = nextChar();

        if (c == ' ' || c == '\n' || c == '\t' || c == '\r') {
            startLexema();
            continue;
        }

        if (c == '\0') {
            if (first == current) {
                return new Token(Token::END);
            } else {
                return new Token(Token::ERR, "Unexpected end of input");
            }
        }

        if (c == '(') return new Token(Token::LPAREN);
        if (c == ')') return new Token(Token::RPAREN);
        if (c == '+') return new Token(Token::PLUS, c);
        if (c == '-') return new Token(Token::MINUS, c);
        if (c == ';') return new Token(Token::SEMICOLON, c);

        if (c == '*') {
            char next = nextChar();
            if (next == '*') {
                return new Token(Token::POW, "**");
            } else {
                rollBack();
                return new Token(Token::MULT, '*');
            }
```

```
        }

        if (c == '/') return new Token(Token::DIV, c);

        if (isalpha(c)) {
            string id;
            id += c;
            while ((c = nextChar()) && (isalnum(c))) {
                id += c;
            }
            rollBack();

            if (id == "sin") return new Token(Token::SIN);
            if (id == "cos") return new Token(Token::COS);
            if (id == "log") return new Token(Token::LOG);
            if (id == "pi") return new Token(Token::PI);
            if (id == "e") return new Token(Token::E);
            return new Token(Token::ID, id);
        }

        if (isdigit(c)) {
            string num;
            num += c;
            bool hasDecimal = false;

            while ((c = nextChar()) && (isdigit(c) || (c == '.' && !hasDecimal))) {
                if (c == '.') hasDecimal = true;
                num += c;
            }

            if (c == ';') {
                rollBack();
                Token* numToken = new Token(Token::NUM, num);
                startLexema();
                return numToken;
            }
            rollBack();
            return new Token(Token::NUM, num);
        }

        return new Token(Token::ERR, string(1, c));
    }
}

Token* Scanner::nextTokenWithLine() {
    char c;
    startLexema();

    while (true) {
        c = nextChar();

        if (c == ' ' || c == '\t' || c == '\r') {
            startLexema();
            continue;
```

```
158         }
159 
160         if (c == '\n') {
161             currentLine++;
162             startLexema();
163             continue;
164         }
165 
166         if (c == '\0') {
167             if (first == current) {
168                 tokensTable.push_back(new Token(Token::END, currentLine));
169                 return tokensTable.back();
170             } else {
171                 tokensTable.push_back(new Token(Token::ERR, "Unexpected end of input",
    currentLine));
172                 return tokensTable.back();
173             }
174         }
175 
176         if (c == '(') {
177             tokensTable.push_back(new Token(Token::LPAREN,"(" ,currentLine));
178             return tokensTable.back();
179         }
180         if (c == ')') {
181             tokensTable.push_back(new Token(Token::RPAREN, ")",currentLine));
182             return tokensTable.back();
183         }
184         if (c == '+') {
185             tokensTable.push_back(new Token(Token::PLUS, c, currentLine));
186             return tokensTable.back();
187         }
188         if (c == '-') {
189             tokensTable.push_back(new Token(Token::MINUS, c, currentLine));
190             return tokensTable.back();
191         }
192         if (c == ';') {
193             tokensTable.push_back(new Token(Token::SEMICOLON, c, currentLine));
194             return tokensTable.back();
195         }
196 
197         if (c == '*') {
198             char next = nextChar();
199             if (next == '*') {
200                 tokensTable.push_back(new Token(Token::POW, "**", currentLine));
201                 return tokensTable.back();
202             } else {
203                 rollBack();
204                 tokensTable.push_back(new Token(Token::MULT, '*', currentLine));
205                 return tokensTable.back();
206             }
207         }
208 
209         if (c == '/') {
210             tokensTable.push_back(new Token(Token::DIV, c, currentLine));
```

```
211             return tokensTable.back();
212         }
213 
214         if (isalpha(c)) {
215             string id;
216             id += c;
217             while ((c = nextChar()) && (isalnum(c))) {
218                 id += c;
219             }
220             rollBack();
221 
222             if (id == "sin") {
223                 tokensTable.push_back(new Token(Token::SIN,"sin", currentLine));
224                 return tokensTable.back();
225             }
226             if (id == "cos") {
227                 tokensTable.push_back(new Token(Token::COS,"cos", currentLine));
228                 return tokensTable.back();
229             }
230             if (id == "log") {
231                 tokensTable.push_back(new Token(Token::LOG,"log", currentLine));
232                 return tokensTable.back();
233             }
234             if (id == "pi") {
235                 tokensTable.push_back(new Token(Token::PI,"pi", currentLine));
236                 return tokensTable.back();
237             }
238             if (id == "e") {
239                 tokensTable.push_back(new Token(Token::E,"e", currentLine));
240                 return tokensTable.back();
241             }
242             tokensTable.push_back(new Token(Token::ID, id, currentLine));
243             return tokensTable.back();
244         }
245 
246         if (isdigit(c)) {
247             string num;
248             num += c;
249             bool hasDecimal = false;
250 
251             while ((c = nextChar()) && (isdigit(c) || (c == '.' && !hasDecimal))) {
252                 if (c == '.') hasDecimal = true;
253                 num += c;
254             }
255 
256             if (c == ';') {
257                 rollBack();
258                 tokensTable.push_back(new Token(Token::NUM, num, currentLine));
259                 return tokensTable.back();
260             }
261             rollBack();
262             tokensTable.push_back(new Token(Token::NUM, num, currentLine));
263             return tokensTable.back();
264         }
```

```
265
266         tokensTable.push_back(new Token(Token::ERR, string(1, c), currentLine));
267         return tokensTable.back();
268     }
269 }
270
271 void Scanner::printTokenTable() {
272     cout << "+------------+------------+-------+" << endl;
273     cout << "| Lexema     | Token      | Linea |" << endl;
274     cout << "+------------+------------+-------+" << endl;
275
276     for (Token* token : tokensTable) {
277         if (token->type == Token::END) continue;
278
279         cout << "| " << left << setw(10) << token->lexema << " | "
280              << setw(10) << Token::token_names[token->type] << " | "
281              << setw(5) << token->line << " |" << endl;
282     }
283
284     cout << "+------------+------------+-------+" << endl;
285 }
286
287 Scanner::~Scanner() {
288     for (Token* token : tokensTable) {
289         delete token;
290     }
291 }
292
293 char Scanner::nextChar() {
294     int c = input[current];
295     if (c != '\0') current++;
296     return c;
297 }
298
299 void Scanner::rollBack() {
300     if (current > 0)
301         current--;
302 }
303
304 void Scanner::startLexema() {
305     first = current;
306 }
307
308 string Scanner::getLexema() {
309     return input.substr(first, current-first);
310 }
311
312 int main(int argc, const char* argv[]) {
313     if (argc != 2) {
314         cout << "Uso: " << argv[0] << " <archivo>" << endl;
315         return 1;
316     }
317
318
```

```
    ifstream file(argv[1]);
    if (!file) {
        cout << "Error al abrir el archivo " << argv[1] << endl;
        return 1;
    }

    string input((istreambuf_iterator<char>(file)), istreambuf_iterator<char>());
    file.close();

    if (input.empty() || input.back() != ';') {
        cout << "Error: El archivo debe terminar con un punto y coma (;)" << endl;
        return 1;
    }

    // Mostrar salida original
    cout << "=== SALIDA SCANNER ===" << endl;
    Scanner scanner1(input.c_str());
    Token* tk = scanner1.nextToken();
    while (tk->type != Token::END) {
        cout << "next token " << tk << endl;
        delete tk;
        tk = scanner1.nextToken();
    }
    cout << "last token " << tk << endl;
    delete tk;

    // Mostrar tabla de tokens
    cout << "\n=== TABLA DE TOKENS ===" << endl;
    Scanner scanner2(input.c_str());
    tk = scanner2.nextTokenWithLine();
    while (tk->type != Token::END) {
        tk = scanner2.nextTokenWithLine();
    }
    scanner2.printTokenTable();
    // cd ".\Compiladores\"
    // g++ lab4_plantilla.cpp -o lab4_plantilla.exe
    // .\'lab4_plantilla.exe' input.txt
    /*
        La entrada se encuentra en el archivo input.txt y
        debe terminar con un punto y coma (;).

        El programa escanea el archivo y muestra los tokens
        encontrados, así como una tabla de tokens.
    */
    return 0;
}
```