```cpp
#include "Lexer.h"


Lexer::Lexer() : comment(false) {}

Lexer::~Lexer() {}

std::vector<Lexer::Token> Lexer::lex(std::stringstream &buffer, int lineNumber)
{
    std::vector<Token> tokens;
    Token *token;
    char c;
    int transition;
    std::string lexeme = "";
    std::string tokenStr = "";
    int prevState = 0;
    int currState = 0;

    while (buffer.get(c))
    {
        // Check if we are inside of a multiline comment,
        // or at the beginning of a new comment range.
        if (comment | (c == '[' && buffer.peek() == '*'))
        {
            // Iterate until we see a '*]'
            while (c != '*' | buffer.peek() != ']')
            {
                // If we hit the end of the line, set
                // comment switch to "true" and reset
                // the current character so it gets ignored.
                if (buffer.eof())
                {
                    comment = true;
                    c = ' ';
                    break;
                }

                buffer.get(c);
            }

            // If we haven't reached the end of the file,
            // and the current character is a '*', we know
            // we have reached the end of the comment section.
            if (!buffer.eof() && c == '*')
            {
                comment = false;

                // Get both characters '*]' out of the stream
                buffer.get(c).get(c);
            }
        }

        // Get the character type (transition)
        transition = getTransition(c);

        // Update state
        currState = Lexer::stateTable[currState][transition];

        // Terminating state
        if (currState == TRM)
        {
            tokenStr = stateToString(prevState);

            if (tokenStr != "Illegal")
            {

                if (tokenStr == "Identifier")
                {
                    // Check if this identifier is a keyword
                    if (isKeyword(lexeme))
```

```
71:                     {
72:                         tokenStr = "Keyword";
73:                     }
74:                 }
75:
76:                 // Create token and add to list of tokens
77:                 token = new Token(tokenStr, lexeme, lineNumber);
78:                 tokens.push_back(*token);
79:
80:                 // reset state machine
81:                 currState = NS;
82:                 lexeme.clear();
83:                 tokenStr.clear();
84:
85:                 // If we reached the terminating state by anything other
86:                 // than whitespace, we need to put it back and re-examine
87:                 // the character on the next iteration.
88:                 if (!isspace(c))
89:                 {
90:                     buffer.putback(c);
91:                 }
92:             }
93:             else
94:             {
95:                 // Push back rejected token
96:                 if (!lexeme.empty())
97:                 {
98:                     token = new Token(tokenStr, lexeme, lineNumber);
99:                     tokens.push_back(*token);
100:                }
101:
102:                // reset state machine
103:                currState = NS;
104:                lexeme.clear();
105:                tokenStr.clear();
106:            }
107:        }
108:        else
109:        {
110:            if (!isspace(c))
111:            {
112:                lexeme.push_back(c);
113:            }
114:        }
115:
116:        prevState = currState;
117:    }
118:
119:    // Grab the last token
120:    tokenStr = stateToString(prevState);
121:
122:    // Evaluate the last token
123:    if (tokenStr != "Illegal")
124:    {
125:        if (tokenStr == "Identifier")
126:        {
127:            // Check if this identifier is a keyword
128:            if (isKeyword(lexeme))
129:            {
130:                tokenStr = "Keyword";
131:            }
132:        }
133:
134:        // Create token and add to list of tokens
135:        token = new Token(tokenStr, lexeme, lineNumber);
136:        tokens.push_back(*token);
137:    }
138:
139:    return tokens;
140: }
```

```
141:
142: int Lexer::getTransition(char c) const
143: {
144:     int transition = REJECT;
145:
146:     if (isdigit(c))
147:     {
148:         transition = INTEGER;
149:     }
150:     else if (isalpha(c))
151:     {
152:         transition = IDENTIFIER;
153:     }
154:     else if (c == '.')
155:     {
156:         transition = REAL;
157:     }
158:     else if (c == '^')
159:     {
160:         transition = CARROT;
161:     }
162:     else if (c == '=')
163:     {
164:         transition = EQUALS;
165:     }
166:     else if (c == '>')
167:     {
168:         transition = GREATERTHAN;
169:     }
170:     else if (c == '<')
171:     {
172:         transition = LESSTHAN;
173:     }
174:     else if (c == '+')
175:     {
176:         transition = PLUS;
177:     }
178:     else if (c == '-')
179:     {
180:         transition = MINUS;
181:     }
182:     else if (c == '*')
183:     {
184:         transition = MULTIPLY;
185:     }
186:     else if (c == '/')
187:     {
188:         transition = DIVIDE;
189:     }
190:     else if (isValidSeparator(c))
191:     {
192:         transition = SEPARATOR;
193:     }
194:     else if (c == '$')
195:     {
196:         transition = FUNC_SEPARATOR;
197:     }
198:
199:     return transition;
200: }
201:
202: std::string Lexer::stateToString(int state) const
203: {
204:     std::string stateStr = "Illegal";
205:
206:     switch (state)
207:     {
208:     case S01:
209:     case S02:
210:         stateStr = "Identifier";
```

```
211:         break;
212:     case S04:
213:         stateStr = "Integer";
214:         break;
215:     case S06:
216:         stateStr = "Real";
217:         break;
218:     case S07:
219:     case S09:
220:         stateStr = "Separator";
221:         break;
222:     case S11:
223:     case S12:
224:     case S13:
225:     case S14:
226:         stateStr = "Operator";
227:         break;
228:     }
229:
230:     return stateStr;
231: }
232:
233: bool Lexer::isValidSeparator(char c) const
234: {
235:     return separators.count(c);
236: }
237:
238: bool Lexer::isKeyword(std::string token) const
239: {
240:     return keywords.count(token);
241: }
```