```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.Data;
using System.Dynamic;
using System.IO;
using System.Linq;
using System.Runtime.InteropServices.ComTypes;
using System.Text;

namespace KyleBushCompiler
{
    class LexicalAnalyzer
    {
        /// <summary>
        /// Contains all possible states from the DFA diagram.
        /// </summary>
        enum State
        {
            START,
            INTEGER_START,
            INTEGER_ACCEPT,
            FLOATING_POINT_START,
            FLOATING_POINT_SCI_NOTATION,
            FLOATING_POINT_SCI_NOTATION_SIGN,
            FLOATING_POINT_SCI_NOTATION_DIGIT,
            FLOATING_POINT_FRACTIONAL_DIGIT,
            FLOATING_POINT_ACCEPT,
            IDENTIFIER_START,
            IDENTIFIER_ACCEPT,
            STRING_START,
            STRING_ACCEPT,
            COMMENT_2_START,
            COMMENT_2_BODY,
            COMMENT_2_CLOSE,
            COMMENT_1_BODY,
            ONE_OR_TWO_CHAR_TOKEN_ACCEPT,
            UNDEFINED
        }

        private State CurrentState;
        private const int IDENTIFIER = 50;
        private const int INTEGER = 51;
        private const int FLOATING_POINT = 52;
        private const int STRING = 53;
        private const int UNDEFINED = 99;

        private const int MAX_IDENTIFIER_LENGTH = 30;
        private const int MAX_NUMERIC_LENGTH = 16;

        public string NextToken { get; set; }
        public int TokenCode { get; set; }
        public SymbolTable SymbolTable { get; private set; }
        public ReserveTable ReserveTable { get; private set; }
        public bool EndOfFile { get; set; }
        public string[] FileText { get; set; }
        public string CurrentLine { get; set; }
        public char CurrentChar { get; set; }
        public char NextChar { get; set; }
        public int CurrentLineIndex { get; set; }
        public int CurrentCharIndex { get; set; }
        public bool TokenFound { get; set; }
        public bool EchoOn { get; set; }
        public bool EndOfLine { get; private set; }

        /// <summary>
        /// Initializes the Lexical Analyzer to a baseline state.
        /// </summary>
        /// <param name="fileText">The source text as a string array</param>
        /// <param name="symbolTable">The table that will hold all symbols found</param>
        /// <param name="reserveTable">The table containing the reserve words for the langauge</param>
        public void Initialize(string[] fileText, SymbolTable symbolTable, ReserveTable reserveTable)
        {
            SymbolTable = symbolTable;
            ReserveTable = reserveTable;
            EndOfFile = false;
            EchoOn = false;
            FileText = fileText;
            CurrentLineIndex = 0;
            CurrentCharIndex = 0;
            CurrentLine = FileText[CurrentLineIndex];
        }

        /// <summary>
        /// Identifies and returns the next available token in the source code.
        /// </summary>
        /// <param name="echoOn">Selects whether input lines are echoed when read</param>
        public void GetNextToken(bool echoOn)
        {
            CurrentState = State.START;
            EchoOn = echoOn;
            NextToken = "";
            TokenFound = false;

            while (!EndOfFile && !TokenFound)
            {
                GetNextChar();
                // Check for single character comment identifier
                if (CurrentChar == '{')
                {
                    CommentStyleOne();
                }

```

```csharp
                else if (CurrentChar == '(' && LookAhead() == '*')
                {
                    CommentStyleTwo();
                }
                // Check for one or two char tokens
                else if (IsOneOrTwoCharTokenStart(CurrentChar))
                {
                    GetOneOrTwoCharToken(CurrentChar);
                }
                // Check if NUMERIC CONSTANT either INTEGER or FLOATING_POINT
                else if (IsDigit(CurrentChar))
                {
                    GetNumericToken();
                }
                // Check if IDENTIFIER
                else if (IsLetter(CurrentChar))
                {
                    GetIdentifierToken();
                }
                // Check if STRING
                else if (CurrentChar == '"')
                {
                    GetStringToken();
                }
                // Found an undefined character
                else
                {
                    AddCharToNextToken();
                    AcceptToken(UNDEFINED, State.UNDEFINED);
                }
            }

            if (EndOfFile)
            {
                CheckForEndOfFileErrors();
            }
        }

        /// <summary>
        /// Checks if the end of the file was reached before a comment or string was closed.
        /// </summary>
        private void CheckForEndOfFileErrors()
        {
            switch (CurrentState)
            {
                case State.COMMENT_1_BODY:
                case State.COMMENT_2_START:
                case State.COMMENT_2_BODY:
                case State.COMMENT_2_CLOSE:
                    Console.WriteLine("\tWARNING: End of file found before comment terminated");
                    break;
                case State.STRING_START:
                    Console.WriteLine("\tWARNING: Unterminated string found");
            }
        }

        /// <summary>
        /// A string token has been detected. This method will continue to add characters to the
        /// token until the end of the token or end of line is found.
        /// </summary>
        private void GetStringToken()
        {
            CurrentState = State.STRING_START;
            NextChar = LookAhead();
            while (!EndOfFile && NextChar != '"')
            {
                GetNextChar();
                if (EndOfLine)
                {
                    Console.WriteLine("\tWARNING: End of line was reached before \" was found to close string.");
                    break;
                }
                AddCharToNextToken();
                NextChar = LookAhead();
            }

            AcceptToken(STRING, State.STRING_ACCEPT);
            AddTokenToSymbolTable();

            if (NextChar == '"')
                GetNextChar();
        }

        /// <summary>
        /// An identifier has been detected. This method will continue to add characters to the token
        /// until the end of the token is found.
        /// </summary>
        private void GetIdentifierToken()
        {
            CurrentState = State.IDENTIFIER_START;
            AddCharToNextToken();
            while (!EndOfFile && !IsWhitespace(LookAhead()) && IsLetter(LookAhead()) || IsDigit(LookAhead()) || LookAhead() == '_' || LookAhead() == '$')
            {
                GetNextChar();
                AddCharToNextToken();
            }
            AcceptToken(GetIdentifierCode(), State.IDENTIFIER_ACCEPT);
            if (TokenCode == IDENTIFIER)
                AddTokenToSymbolTable();
        }

        /// <summary>
        /// A numeric token has been detected. This determines if the token is an
        /// integer or floating point token and builds that token.
        /// </summary>
        private void GetNumericToken()
        {
            CurrentState = State.INTEGER_START;
            AddCharToNextToken();

            NextChar = LookAhead();

            while (!EndOfFile && IsDigit(NextChar))
            {
                GetNextChar();
                AddCharToNextToken();
                NextChar = LookAhead();
                if (EndOfLine)
                    break;
            }
            if (NextChar == '.')
            {
                GenerateFloatingPointToken();
            }
            else
            {
                AcceptToken(INTEGER, State.INTEGER_ACCEPT);
                AddTokenToSymbolTable();
            }
        }

        /// <summary>
        /// A floating point token has been detected. This method will build that
          token.
        /// </summary>
        private void GenerateFloatingPointToken()
        {
            CurrentState = State.FLOATING_POINT_START;
            GetNextChar();
            AddCharToNextToken();

            NextChar = LookAhead();
            if (IsDigit(NextChar))
            {
                while (!EndOfFile && IsDigit(NextChar))
                {
                    GetNextChar();
                    AddCharToNextToken();
                    NextChar = LookAhead();
                    if (EndOfLine)
                        break;
                }
                if (NextChar == 'E')
                {
                    GenerateFloatingPointScientificNotationToken();
                }
            }
            else if (NextChar == 'E')
            {
                GenerateFloatingPointScientificNotationToken();
            }

            AcceptToken(FLOATING_POINT, State.FLOATING_POINT_ACCEPT);
            AddTokenToSymbolTable();
        }

        /// <summary>
        /// A floating point token using scientific notation has been detected.
        /// This method builds that token.
        /// </summary>
        private void GenerateFloatingPointScientificNotationToken()
        {
            CurrentState = State.FLOATING_POINT_SCI_NOTATION;
            GetNextChar();
            AddCharToNextToken();
            NextChar = LookAhead();

            if (NextChar == '-' || NextChar == '+')
            {
                CurrentState = State.FLOATING_POINT_SCI_NOTATION_SIGN;
                GetNextChar();
                AddCharToNextToken();
                NextChar = LookAhead();
            }

            if (IsDigit(NextChar))
            {
                CurrentState = State.FLOATING_POINT_SCI_NOTATION_DIGIT;
                GetNextChar();
                AddCharToNextToken();
                NextChar = LookAhead();

                while (!EndOfFile && IsDigit(NextChar))
                {
                    GetNextChar();
                    AddCharToNextToken();
                    NextChar = LookAhead();
                    if (EndOfLine)
                        break;
                }
                AcceptToken(FLOATING_POINT, State.FLOATING_POINT_ACCEPT);
                AddTokenToSymbolTable();
            }
            else
            {
                Console.WriteLine("ERROR: Expected at least one digit.");
            }
        }

        /// <summary>
        /// Flags that a token has been found, sets the current state of the DFA,
        /// sets the correct token code, and truncates the token if needed.
        /// </summary>
        /// <param name="tokenCode">The token code of the token that was found</param>
        /// <param name="state">The current state of the DFA</param>
        private void AcceptToken(int tokenCode, State state)
        {
            TokenFound = true;
            CurrentState = state;
            TokenCode = tokenCode;
            TruncateTokenIfTooLong();
        }

        /// <summary>
        /// A comment has been detected using the delimiter (*.
        /// This method ignores all characters until a closing delimiter
        /// or the end of the file is found.
        /// </summary>
        private void CommentStyleTwo()
        {
            CurrentState = State.COMMENT_2_BODY;
            GetNextChar();
            GetNextChar();
            NextChar = LookAhead();

            // TODO: This still exits too early because seeing * causes exit even if NextChar is not )
            while (!EndOfFile && (CurrentChar != '*' && NextChar != ')') || (CurrentChar == '*' && NextChar != ')') || (CurrentChar != '*' && NextChar == ')'))
            {
                GetNextChar();
                NextChar = LookAhead();
            };

            GetNextChar();

            if (!EndOfFile)
                CurrentState = State.START;
        }

        /// <summary>
        /// A comment has been detected using the { delimiter.
        /// This method ignores all characters until a closing delimiter
        /// or the end of the file is found.
        /// </summary>
        private void CommentStyleOne()
        {
            CurrentState = State.COMMENT_1_BODY;
            while (CurrentChar != '}')
            {
                GetNextChar();
                if (EndOfFile)
                    return;
            };

            if (!EndOfFile)
                CurrentState = State.START;
        }

        /// <summary>
        /// Truncates the token if it is too long for the defined token type
        /// and displays a warning message.
        /// </summary>
        private void TruncateTokenIfTooLong()
        {
            // TODO: differentiate between numeric and identifiers.
            int maxLength;

            if (TokenCode == IDENTIFIER)
                maxLength = MAX_IDENTIFIER_LENGTH;
            else if (TokenCode == FLOATING_POINT || TokenCode == INTEGER)
                maxLength = MAX_NUMERIC_LENGTH;
            else
                return;

            if (NextToken.Length > maxLength)
            {
                Console.WriteLine("\tWARNING: Token length exceeds " + maxLength + ". Token has been truncated.");
                NextToken = NextToken.Substring(0, maxLength);
            }
        }

        /// <summary>
        /// Determines if a token is one of the predefined one or two character tokens
        /// from section 6 of the CS4100projectlangFA20-TOKENS.pdf
        /// </summary>
        /// <param name="c">The character being tested.</param>
        /// <returns>True if character is one or two char token. False if not.</returns>
        private bool IsOneOrTwoCharTokenStart(char c)
        {
            switch(c)
            {
                case '/':
                case '*':
                case '+':
                case '-':
                case '(':
                case ')':
                case ';':
                case '=':
                case ',':
                case '[':
                case ']':
                case '.':
                case ':':
                case '>':
```

```
                case '<':
                    return true;
                default:
                    return false;
            }
        }

        /// <summary>
        /// One of the predefined one or two character tokens
        /// from section 6 of the CS4100projectlangFA20-TOKENS.pdf
        /// has been detected so this method stores it in NextToken.
        /// </summary>
        /// <param name="c">The current character</param>
        private void GetOneOrTwoCharToken(char c)
        {
            CurrentState = State.ONE_OR_TWO_CHAR_TOKEN_ACCEPT;
            switch (c)
            {
                case '/':
                case '*':
                case '+':
                case '-':
                case '(':
                case ')':
                case ';':
                case '=':
                case ',':
                case '[':
                case ']':
                case '.':
                    NextToken += CurrentChar;
                    break;
                case ':':
                    if (LookAhead() == '=')
                    {
                        NextToken += CurrentChar;
                        GetNextChar();
                        NextToken += CurrentChar;
                    }
                    else
                    {
                        NextToken += CurrentChar;
                    }
                    break;
                case '>':
                    if (LookAhead() == '=')
                    {
                        NextToken += CurrentChar;
                        GetNextChar();
                        NextToken += CurrentChar;
                    }
                    else
                    {
                        NextToken += CurrentChar;
                    }
                    break;
                case '<':
                    if (LookAhead() == '=' || LookAhead() == '>')
                    {
                        NextToken += CurrentChar;
                        GetNextChar();
                        NextToken += CurrentChar;
                    }
                    else
                    {
                        NextToken += CurrentChar;
                    }
                    break;
            }
            AcceptToken(ReserveTable.LookupName(NextToken),
              State.ONE_OR_TWO_CHAR_TOKEN_ACCEPT);
        }

        /// <summary>
        /// Peeks at the next character without advancing.
        /// </summary>
        /// <returns>The next character without advancing.</returns>
        private char LookAhead()
        {
            char lookAhead = ' ';
            if (CurrentCharIndex < CurrentLine.Length)
            {
                lookAhead = CurrentLine[CurrentCharIndex];
            }
            return lookAhead;
        }

        /// <summary>
        /// Checks if the token is already in the symbol table.
        /// If it is not then it is added, otherwise it does nothing.
        /// </summary>
        private void AddTokenToSymbolTable()
        {
            string tokenToAdd;
            if (TokenCode == IDENTIFIER)
                tokenToAdd = NextToken.ToUpper();
            else
                tokenToAdd = NextToken;

            int symbolIndex = SymbolTable.LookupSymbol(tokenToAdd);
            if (symbolIndex == -1)
            {
                switch (TokenCode)
                {
                    case IDENTIFIER:
                        SymbolTable.AddSymbol(tokenToAdd, SymbolKind.Variable,
                    0);
                        break;
                    case INTEGER:
                        SymbolTable.AddSymbol(tokenToAdd, SymbolKind.Constant, 
                    Int64.Parse(tokenToAdd));
                        break;
                    case FLOATING_POINT:
                        SymbolTable.AddSymbol(tokenToAdd, SymbolKind.Constant, 
                    Double.Parse(tokenToAdd));
                        break;
                    case STRING:
                        SymbolTable.AddSymbol(tokenToAdd, SymbolKind.Constant, 
                    tokenToAdd);
                        break;
                }
            }
        }

        /// <summary>
        /// Queries the Reserve Table to determine if the current token is a 
          reserve word.
        /// If it is then the proper token code is returned from the table.
        /// If it is not a reserve word it is given the identifier token code.
        /// </summary>
        /// <returns></returns>
        private int GetIdentifierCode()
        {
            int code = ReserveTable.LookupName(NextToken.ToUpper());
            if (code == -1)
            {
                return IDENTIFIER;
            }

            return code;
        }

        /// <summary>
        /// Adds the current char to NextToken.
        /// </summary>
        private void AddCharToNextToken()
        {
            NextToken += CurrentChar;
        }

        /// <summary>
        /// Get's the next line of source text and prints it if EchoOn is true
        /// </summary>
        private void GetNextLine()
        {
            if (CurrentLineIndex < FileText.Length)
            {
                CurrentLine = FileText[CurrentLineIndex];
                CurrentLineIndex++;
            }

            if (EchoOn)
            {
                Console.WriteLine(CurrentLine);
            }
        }

        /// <summary>
        /// Get's the next character from the source text.
        /// Also, checks for the end of the file and the end of a line.
        /// Skips blanks that are not part of a token.
        /// </summary>
        private void GetNextChar()
        {
            if (IsEndOfFile())
            {
                EndOfFile = true;
                return;
            }

            if (IsEndOfLine())
            {
                if (IsCommentOrStart())
                {
                    GetNextLine();
                    CurrentCharIndex = 0;
                    EndOfLine = false;
                }
                else
                {
                    EndOfLine = true;
                    return;
                }
            }

            if (!string.IsNullOrEmpty(CurrentLine))
            {
                CurrentChar = CurrentLine[CurrentCharIndex];
                CurrentCharIndex++;
            }

            if (CurrentState == State.START)
            {
                SkipBlanks();
            }
        }

        /// <summary>
        /// Determines if the current state of the DFA is a comment or start.
        /// </summary>
        /// <returns>True if the DFA is in a comment of start state. False if 
          not.</returns>
        private bool IsCommentOrStart()
        {
            switch (CurrentState)
            {
                case State.START:
                case State.COMMENT_1_BODY:
                case State.COMMENT_2_START:
                case State.COMMENT_2_BODY:
```

```
                case State.COMMENT_2_CLOSE:
                    return true;
            }
            return false;
        }

        /// <summary>
        /// Skips blanks and empty lines that are not part of tokens.
        /// </summary>
        private void SkipBlanks()
        {
            while (!EndOfFile && IsWhitespace(CurrentChar) ||
              string.IsNullOrEmpty(CurrentLine))
            {
                GetNextChar();
            }
        }

        /// <summary>
        /// Checks if the end of the file has been found.
        /// </summary>
        /// <returns>True if end of line is found. False if not.</returns>
        private bool IsEndOfFile()
        {
            return (CurrentLineIndex == FileText.Length && CurrentCharIndex ==
              CurrentLine.Length);
        }

        /// <summary>
        /// Checks if the end of a line has been found.
        /// </summary>
        /// <returns>True if end of line is found. False if not</returns>
        private bool IsEndOfLine()
        {
            return CurrentCharIndex == CurrentLine.Length;
        }

        /// <summary>
        /// Checks if a character is a letter.
        /// </summary>
        /// <param name="c"></param>
        /// <returns>True if char is letter. False if not.</returns>
        private bool IsLetter(char c)
        {
            return (c >= 'A' && c <= 'Z') || (c >= 'a' && c <= 'z');
        }

        /// <summary>
        /// Checks if a character is a digit.
        /// </summary>
        /// <param name="c"></param>
        /// <returns>True if char is digit. False if not.</returns>
        private bool IsDigit(char c)
        {
            return (c >= '0' && c <= '9');
        }

        /// <summary>
        /// Checks if a character is whitespace.
        /// </summary>
        /// <param name="c"></param>
        /// <returns>True if char is whitespace. False if not.</returns>
        private bool IsWhitespace(char c)
        {
            return char.IsWhiteSpace(c);
        }
    }
}
```