```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.Design;
using System.IO;
using System.Reflection.Emit;

namespace KyleBushCompiler
{
    class Program
    {
        /*
         * CFG for Language Definition
         * <program> -> $UNIT <prog-identifier> $SEMICOLON <block> $PERIOD
         * <block> -> $BEGIN <statement> {$SEMICOLON <statement>}* $END
         * <prog-identifier> -> <identifier>
         * <statement> -> <variable> $COLON_EQUALS <simple expression>
         * <variable> -> <identifier>
         * <simple expression> -> [<sign>] <term> {<addop> <term>}*
         * <addop> -> $PLUS | $MINUS
         * <sign> -> $PLUS | $MINUS
         * <term> -> <factor> {<mulop> <factor> }*
         * <mulop> -> $MULTIPLY | $DIVIDE
         * <factor> -> <unsigned constant> | <variable> | $LPAR <simple expression> $RPAR
         * <unsigned constant>-> <unsigned number>
         * <unsigned number>-> $FLOAT | $INTTYPE
         * <identifier> -> $IDENTIFIER
         */
        static void Main(string[] args)
        {
            // Provided GOOD test file
            string inputFilePath = @"C:\projects\CS4100_Compiler_Design\TestInput\GoodtreeA.txt";

            // Provided BAD test file with syntax error
            // string inputFilePath = @"C:\projects\CS4100_Compiler_Design\TestInput\BadProg1.txt";

            // Provided BAD test file with lexical and syntax error
            // string inputFilePath = @"C:\projects\CS4100_Compiler_Design\TestInput\BadProg2.txt";

            // Initialize structures
            ReserveTable reserveWords = InitializeReserveWordTable();
            ReserveTable tokenCodes = InitializeTokenCodeTable();
            SymbolTable symbolTable = new SymbolTable();

            try
            {
                // Initialize input file
                string[] fileText = InitializeInputFile(inputFilePath);

                // Initialize the Lexical Analyzer (Scanner)
                LexicalAnalyzer scanner = new LexicalAnalyzer();

                scanner.Initialize(fileText, symbolTable, reserveWords);
                bool echoOn = true;

                SyntaxAnalyzer parser = new SyntaxAnalyzer(scanner, tokenCodes, echoOn);

                scanner.GetNextToken(echoOn);
                parser.TraceOn = true;
                int val = parser.Program();

                symbolTable.PrintSymbolTable();
            }
            catch (Exception e)
            {
                Console.WriteLine(e.Message);
            }
        }

        /// <summary>
        /// Initializes the reserve table containing the token codes and mnemonics
        /// </summary>
        /// <returns>Reserve table containing the token codes and mnemonics</returns>
        static ReserveTable InitializeTokenCodeTable()
        {
            ReserveTable tokenCodes = new ReserveTable();

            // Reserve Words
            tokenCodes.Add("GOTO", 0);
            tokenCodes.Add("_INT", 1);
            tokenCodes.Add("__TO", 2);
            tokenCodes.Add("__DO", 3);
            tokenCodes.Add("__IF", 4);
            tokenCodes.Add("THEN", 5);
            tokenCodes.Add("ELSE", 6);
            tokenCodes.Add("_FOR", 7);
            tokenCodes.Add("__OF", 8);
            tokenCodes.Add("WTLN", 9);
            tokenCodes.Add("RDLN", 10);
            tokenCodes.Add("_BEG", 11);
            tokenCodes.Add("_END", 12);
            tokenCodes.Add("_VAR", 13);
            tokenCodes.Add("WHIL", 14);
            tokenCodes.Add("UNIT", 15);
            tokenCodes.Add("LABL", 16);
            tokenCodes.Add("REPT", 17);
            tokenCodes.Add("UNTL", 18);
            tokenCodes.Add("PROC", 19);
            tokenCodes.Add("DOWN", 20);
            tokenCodes.Add("FUNC", 21);
            tokenCodes.Add("RTRN", 22);
            tokenCodes.Add("REAL", 23);
            tokenCodes.Add("_STR", 24);
            tokenCodes.Add("ARRY", 25);

            // Other Tokens
            tokenCodes.Add("_DIV", 30);
            tokenCodes.Add("_MUL", 31);
            tokenCodes.Add("_ADD", 32);
            tokenCodes.Add("_SUB", 33);
            tokenCodes.Add("LPAR", 34);
            tokenCodes.Add("RPAR", 35);
            tokenCodes.Add("SEMI", 36);
            tokenCodes.Add("ASGN", 37);
            tokenCodes.Add("__GT", 38);
            tokenCodes.Add("__LT", 39);
            tokenCodes.Add("GTEQ", 40);
            tokenCodes.Add("LTEQ", 41);
            tokenCodes.Add("__EQ", 42);
            tokenCodes.Add("NTEQ", 43);
            tokenCodes.Add("COMM", 44);
            tokenCodes.Add("LBRC", 45);
            tokenCodes.Add("RBRC", 46);
            tokenCodes.Add("COLN", 47);
            tokenCodes.Add("_DOT", 48);

            // Identifiers
            tokenCodes.Add("IDNT", 50);

            // Numeric Constants
            tokenCodes.Add("INTC", 51);
            tokenCodes.Add("FLTC", 52);

            // String
            tokenCodes.Add("STRC", 53);

            // Used for any other input characters which are not defined.
            tokenCodes.Add("UNDF", 99);

            return tokenCodes;
        }

        /// <summary>
        /// Initializes reserve table with reserve words and token codes
        /// </summary>
        /// <returns>Reserve table with reserve words and token codes</returns>
        static ReserveTable InitializeReserveWordTable()
        {
            ReserveTable reserveWords = new ReserveTable();

            // Token Codes
            reserveWords.Add("GOTO", 0);
            reserveWords.Add("INTEGER", 1);
            reserveWords.Add("TO", 2);
```

```
                reserveWords.Add("DO", 3);
                reserveWords.Add("IF", 4);
                reserveWords.Add("THEN", 5);
                reserveWords.Add("ELSE", 6);
                reserveWords.Add("FOR", 7);
                reserveWords.Add("OF", 8);
                reserveWords.Add("WRITELN", 9);
                reserveWords.Add("READLN", 10);
                reserveWords.Add("BEGIN", 11);
                reserveWords.Add("END", 12);
                reserveWords.Add("VAR", 13);
                reserveWords.Add("WHILE", 14);
                reserveWords.Add("UNIT", 15);
                reserveWords.Add("LABEL", 16);
                reserveWords.Add("REPEAT", 17);
                reserveWords.Add("UNTIL", 18);
                reserveWords.Add("PROCEDURE", 19);
                reserveWords.Add("DOWNTO", 20);
                reserveWords.Add("FUNCTION", 21);
                reserveWords.Add("RETURN", 22);
                reserveWords.Add("REAL", 23);
                reserveWords.Add("STRING", 24);
                reserveWords.Add("ARRAY", 25);

                // Other Tokens
                reserveWords.Add("/", 30);
                reserveWords.Add("*", 31);
                reserveWords.Add("+", 32);
                reserveWords.Add("-", 33);
                reserveWords.Add("(", 34);
                reserveWords.Add(")", 35);
                reserveWords.Add(";", 36);
                reserveWords.Add(":=", 37);
                reserveWords.Add(">", 38);
                reserveWords.Add("<", 39);
                reserveWords.Add(">=", 40);
                reserveWords.Add("<=", 41);
```

```
            reserveWords.Add("=", 42);
            reserveWords.Add("<>", 43);
            reserveWords.Add(",", 44);
            reserveWords.Add("[", 45);
            reserveWords.Add("]", 46);
            reserveWords.Add(":", 47);
            reserveWords.Add(".", 48);

            return reserveWords;
        }

        /// <summary>
        /// Reads all the text from the source file and stores each line as a seperate element in a string array.
        /// </summary>
        /// <param name="filePath">Path to the file to be read into memory</param>
        /// <returns>The source text as a string array</returns>
        static string[] InitializeInputFile(string filePath)
        {
            return File.ReadAllLines(filePath);
        }
    }
}
```

```
using System;
using System.Collections.Generic;
using System.ComponentModel.Design;
using System.IO;
using System.Linq.Expressions;
using System.Security.Cryptography.X509Certificates;
using System.Text;

namespace KyleBushCompiler
{
    class SyntaxAnalyzer
    {
        #region Token Constants
        private const int GOTO = 0;
        private const int INTEGER = 1;
        private const int TO = 2;
        private const int DO = 3;
        private const int IF = 4;
        private const int THEN = 5;
        private const int ELSE = 6;
        private const int FOR = 7;
        private const int OF = 8;
        private const int WRITELN = 9;
        private const int READLN = 10;
        private const int BEGIN = 11;
        private const int END = 12;
        private const int VAR = 13;
        private const int WHILE = 14;
        private const int UNIT = 15;
        private const int LABEL = 16;
        private const int REPEAT = 17;
        private const int UNTIL = 18;
        private const int PROCEDURE = 19;
        private const int DOWNTO = 20;
        private const int FUNCTION = 21;
        private const int RETURN = 22;
        private const int REAL = 23;
        private const int STRING = 24;
        private const int ARRAY = 25;
        private const int DIVIDE = 30;
        private const int MULTIPLY = 31;
        private const int PLUS = 32;
        private const int MINUS = 33;
        private const int LPAR = 34;
        private const int RPAR = 35;
        private const int SEMICOLON = 36;
        private const int COLON_EQUALS = 37;
        private const int GREATER_THAN = 38;
        private const int LESS_THAN = 39;
        private const int GREATER_THAN_OR_EQUAL = 40;
        private const int LESS_THAN_OR_EQUAL = 41;
        private const int EQUAL = 42;
        private const int NOT = 43;
        private const int COMMA = 44;
        private const int LEFT_BRACKET = 45;
        private const int RIGHT_BRACKET = 46;
        private const int COLON = 47;
        private const int PERIOD = 48;
        private const int IDENTIFIER = 50;
        private const int INTTYPE = 51;
        private const int FLOAT = 52;
        private const int STRINGTYPE = 53;
        private const int UNDEFINED = 99;
        #endregion

        #region Properties
        public bool TraceOn { get; set; }
        public bool IsError { get; set; }
        private LexicalAnalyzer Scanner { get; set; }
        private ReserveTable TokenCodes { get; set; }
        private bool ScannerEchoOn { get; set; }
        private bool Verbose { get; set; }

        #endregion

        public SyntaxAnalyzer(LexicalAnalyzer scanner, ReserveTable tokenCodes, bool scannerEchoOn)
        {
            Scanner = scanner;
            ScannerEchoOn = scannerEchoOn;
            TokenCodes = tokenCodes;
        }

        #region CFG Methods

        /// <summary>
        /// Implements CFG Rule: <program> -> $UNIT <prog-identifier> $SEMICOLON <block> $PERIOD
        /// </summary>
        /// <returns></returns>
        public int Program()
        {
            if (IsError)
                return -1;

            Debug(true, "Program()");

            if (Scanner.TokenCode == UNIT)
            {
                GetNextToken();
                int x = ProgIdentifier();
                if (Scanner.TokenCode == SEMICOLON)
                {
                    GetNextToken();
                    x = Block();
                    if (Scanner.TokenCode == PERIOD)
                    {
                        GetNextToken();
                    }
                    else
                    {
                        Error("PERIOD");
                    }
                }
                else
```

```
                        Error("SEMICOLON");
                    }
                }
                else
                {
                    Error("UNIT");
                }

                Debug(false, "Program()");
                return -1;
            }

            /// <summary>
            /// Implements CFG Rule: <block> -> $BEGIN <statement> {$SEMICOLON <statement>}* $END
            /// </summary>
            /// <returns></returns>
            private int Block()
            {
                if (IsError)
                    return -1;

                Debug(true, "Block()");
                if (Scanner.TokenCode == BEGIN)
                {
                    GetNextToken();
                    int x = Statement();
                    while (Scanner.TokenCode == SEMICOLON && !IsError)
                    {
                        GetNextToken();
                        x = Statement();
                    }

                    if (Scanner.TokenCode == END)
                        GetNextToken();
                    else
                        Error("END");
                }
                else
                    Error("BEGIN");

                Debug(false, "Block()");
                return -1;
            }

            /// <summary>
            /// Implements CFG Rule: <prog-identifier> -> <identifier>
            /// </summary>
            /// <returns></returns>
            private int ProgIdentifier()
            {
                if (IsError)
                    return -1;

                Debug(true, "ProgIdentifier()");
                Identifier();
                Debug(false, "ProgIdentifier()");
                return -1;
            }

            /// <summary>
            /// Implements CFG Rule: <statement> -> <variable> $COLON_EQUALS <simple expression>
            /// </summary>
            /// <returns></returns>
            private int Statement()
            {
                if (IsError)
                    return -1;

                Debug(true, "Statement()");
                int x = Variable();
                if (Scanner.TokenCode == COLON_EQUALS)
                {
                    GetNextToken();
                    x = SimpleExpression();
                }
                else
                    Error("COLON-EQUALS");

                Debug(false, "Statement()");
```

```
                return -1;
            }

            /// <summary>
            /// Implements CFG Rule: <variable> -> <identifier>
            /// </summary>
            /// <returns></returns>
            private int Variable()
            {
                if (IsError)
                    return -1;

                Debug(true, "Variable()");
                Identifier();
                Debug(false, "Variable()");
                return -1;
            }

            /// <summary>
            /// Implements CFG Rule: <simple expression> -> [<sign>] <term> {<addop> <term>}*
            /// </summary>
            /// <returns></returns>
            private int SimpleExpression()
            {
                if (IsError)
                    return -1;

                Debug(true, "SimpleExpression()");

                int x;

                if (isSign())
                {
                    x = Sign();
                }

                x = Term();

                while (isAddOp() && !IsError)
```

```
                    x = AddOp();
                    x = Term();
                }

                Debug(false, "SimpleExpression()");
                return -1;
            }

            /// <summary>
            /// Implements CFG Rule: <addop> -> $PLUS | $MINUS
            /// </summary>
            /// <returns></returns>
            private int AddOp()
            {
                if (IsError)
                    return -1;

                Debug(true, "AddOp()");
                if (Scanner.TokenCode == PLUS || Scanner.TokenCode == MINUS)
                    GetNextToken();
                else
                    Error("PLUS or MINUS");
                Debug(false, "AddOp()");
                return -1;
            }

            /// <summary>
            /// Checks if the next token is an AddOp token.
            /// </summary>
            /// <returns></returns>
            private bool isAddOp()
            {
                if (Scanner.TokenCode == PLUS || Scanner.TokenCode == MINUS)
                    return true;
                else
                    return false;
            }

            /// <summary>
```

```
        /// Implements CFG Rule: <sign> -> $PLUS | $MINUS
        /// </summary>
        /// <returns></returns>
        private int Sign()
        {
            if (IsError)
                return -1;

            Debug(true, "Sign()");
            if (Scanner.TokenCode == PLUS)
                GetNextToken();
            else if (Scanner.TokenCode == MINUS)
                GetNextToken();
            else
                Error("PLUS or MINUS");
            Debug(false, "Sign()");
            return -1;
        }

        /// <summary>
        /// Checks if the next token is a Sign token.
        /// </summary>
        /// <returns></returns>
        private bool isSign()
        {
            if (Scanner.TokenCode == PLUS || Scanner.TokenCode == MINUS)
                return true;
            else
                return false;
        }

        /// <summary>
        /// Implements CFG Rule: <term> -> <factor> {<mulop> <factor> }*
        /// </summary>
        /// <returns></returns>
        private int Term()
        {
            if (IsError)
```

```
                    return -1;

                Debug(true, "Term()");
                int x = Factor();

                while (isMulOp() && !IsError)
                {
                    x = MulOp();
                    x = Factor();
                }

                Debug(false, "Term()");
                return -1;
            }

            /// <summary>
            /// Implements CFG Rule: <mulop> -> $MULTIPLY | $DIVIDE
            /// </summary>
            /// <returns></returns>
            private int MulOp()
            {
                if (IsError)
                    return -1;

                Debug(true, "MulOp()");

                if (Scanner.TokenCode == MULTIPLY || Scanner.TokenCode == DIVIDE)
                    GetNextToken();
                else
                    Error("MULTIPLY or DIVIDE");

                Debug(false, "MulOp()");
                return -1;
            }

            /// <summary>
            /// Checks if the next token is a MulOp token.
            /// </summary>
            /// <returns></returns>
```

```
        private bool isMulOp()
        {
            if (Scanner.TokenCode == MULTIPLY || Scanner.TokenCode == DIVIDE)
                return true;
            else
                return false;
        }


        /// <summary>
        /// Implements CFG Rule: <factor> -> <unsigned constant> | <variable> | $LPAR <simple expression> $RPAR
        /// </summary>
        /// <returns></returns>
        private int Factor()
        {
            if (IsError)
                return -1;

            Debug(true, "Factor()");

            int x;

            if (isUnsignedConstant())
            {
                x = UnsignedConstant();
            }
            else if (isVariable())
            {
                Variable();
            }
            else if (Scanner.TokenCode == LPAR)
            {
                GetNextToken();
                SimpleExpression();
                if (Scanner.TokenCode == RPAR)
                    GetNextToken();
                else
                    Error("RPAR");
            }
            else
                Error("UNSIGNED CONSTANT or VARIABLE or LPAR");

            Debug(false, "Factor()");
            return -1;
        }



        /// <summary>
        /// Checks if the next token is an Unsigned Constant
        /// </summary>
        /// <returns></returns>
        private bool isUnsignedConstant()
        {
            if (Scanner.TokenCode == FLOAT || Scanner.TokenCode == INTTYPE)
                return true;
            else
                return false;
        }

        /// <summary>
        /// Implements CFG Rule: <unsigned constant>-> <unsigned number>
        /// </summary>
        /// <returns></returns>
        private int UnsignedConstant()
        {
            if (IsError)
                return -1;

            Debug(true, "UnsignedConstant()");
            UnsignedNumber();
            Debug(false, "UnsignedConstant()");
            return -1;
        }

        /// <summary>
        /// Implements CFG Rule: <unsigned number>-> $FLOAT | $INTTYPE
        /// </summary>
        /// <returns></returns>
        private int UnsignedNumber()
        {
            if (IsError)
                return -1;

            Debug(true, "UnsignedNumber()");

            if (Scanner.TokenCode == FLOAT || Scanner.TokenCode == INTTYPE)
                GetNextToken();
            else
                Error("FLOAT or INTTYPE");

            Debug(false, "UnsignedNumber()");
            return -1;
        }

        /// <summary>
        /// Checks if the next token is a Variable
        /// </summary>
        /// <returns></returns>
        private bool isVariable()
        {
            if (Scanner.TokenCode == IDENTIFIER)
                return true;
            else
                return false;
        }

        /// <summary>
        /// Implements CFG Rule: <identifier> -> $IDENTIFIER
        /// </summary>
        /// <returns></returns>
        private int Identifier()
        {
            if (IsError)
                return -1;

            Debug(true, "Identifier()");
```

```csharp

            if (Scanner.TokenCode == IDENTIFIER)
                GetNextToken();
            else
                Error("IDENTIFIER");

            Debug(false, "Identifier()");
            return -1;
        }

        #endregion

        #region Utility Methods

        /// <summary>
        /// Prints an error with the expected token type and the actual token found.
        /// </summary>
        /// <param name="expectedToken">The expected token type.</param>
        private void Error(string expectedToken)
        {
            IsError = true;
            Console.WriteLine("Line #{0}: {1}", Scanner.CurrentLineIndex + 1, Scanner.CurrentLine);
            Console.WriteLine("ERROR: {0} expected, but {1} found.", expectedToken, Scanner.NextToken);
        }


        /// <summary>
        /// Prints the method that is being entered or exited if TraceOn is set to true
        /// </summary>
        /// <param name="entering"></param>
        /// <param name="name"></param>
        private void Debug(bool entering, string name)
        {
            if (TraceOn)
            {
                if (entering)
                    Console.WriteLine("ENTERING " + name);
                else
                    Console.WriteLine("EXITING " + name);
            }
        }

        /// <summary>
        /// Gets the next token and prints the token lexeme and mneumonic if Trace is on.
        /// </summary>
        private void GetNextToken()
        {
            Scanner.GetNextToken(ScannerEchoOn);
            if (TraceOn)
                Console.WriteLine("Lexeme: {0} Mnemonic: {1}", Scanner.NextToken, TokenCodes.LookupCode
                  (Scanner.TokenCode));
        }

        #endregion
    }
}
```