

Mnemonics Listing

Token Name	Token Code	Mnemonic
GOTO	0	GOTO
INTEGER	1	_INT
TO	2	__TO
DO	3	__DO
IF	4	__IF
THEN	5	THEN
ELSE	6	ELSE
FOR	7	_FOR
OF	8	__OF
Writeln	9	WTLN
Readln	10	RDLN
BEGIN	11	_BEG
END	12	_END
VAR	13	_VAR
WHILE	14	WHIL
UNIT	15	UNIT
LABEL	16	LABL
REPEAT	17	REPT
UNTIL	18	UNTL
PROCEDURE	19	PROC
Downto	20	DOWN
FUNCTION	21	FUNC
RETURN	22	RTRN
REAL	23	REAL
STRING	24	_STR
ARRAY	25	ARRY
/	30	_DIV
*	31	_MUL
+	32	_ADD
-	33	_SUB
(34	LPAR
)	35	RPAR
;	36	SEMI
:=	37	ASGN
>	38	__GT
<	39	__LT
>=	40	GTEQ
<=	41	LTEQ
=	42	__EQ
<>	43	NTEQ
,	44	COMM


```
1 using System;
2 using System.Collections.Generic;
3 using System.ComponentModel.Design;
4 using System.IO;
5 using System.Reflection.Emit;
6
7 namespace KyleBushCompiler
8 {
9     class Program
10     {
11         static void Main(string[] args)
12         {
13             // My test file
14             //string inputFilePath = @"C:\projects\CS4100_Compiler_Design \TestInput\program.txt";
15
16             // My test file
17             //string inputFilePath = @"C:\projects\CS4100_Compiler_Design \TestInput\GetNextCharTest.txt";
18
19             // Provided test file
20             string inputFilePath = @"C:\projects\CS4100_Compiler_Design\TestInput \LexicalTestF20.txt";
21
22             // Initialize structures
23             ReserveTable reserveWords = InitializeReserveWordTable();
24             ReserveTable tokenCodes = InitializeTokenCodeTable();
25             SymbolTable symbolTable = new SymbolTable();
26
27             try
28             {
29                 // Initialize input file
30                 string[] fileText = InitializeInputFile(inputFilePath);
31
32                 // Initialize the Lexical Analyzer (Scanner)
33                 LexicalAnalyzer scanner = new LexicalAnalyzer();
34                 scanner.Initialize(fileText, symbolTable, reserveWords);
35                 bool echoOn = true;
36
37                 while (!scanner.EndOfFile)
38                 {
39                     scanner.GetNextToken(echoOn);
40                     if (!scanner.EndOfFile)
41                         PrintToken(scanner.NextToken, scanner.TokenCode,
42                                     tokenCodes, symbolTable);
43
44                     symbolTable.PrintSymbolTable();
45                 }
46             } catch (Exception e)
47             {
48                 Console.WriteLine(e.Message);
49             }
50         }
51     }
```

```
52     /// <summary>
53     /// Initializes the reserve table containing the token codes and mnemonics
54     /// </summary>
55     /// <returns>Reserve table containing the token codes and mnemonics</returns>
56     static ReserveTable InitializeTokenCodeTable()
57     {
58         ReserveTable tokenCodes = new ReserveTable();
59
60         // Reserve Words
61         tokenCodes.Add("GOTO", 0);
62         tokenCodes.Add("_INT", 1);
63         tokenCodes.Add("__TO", 2);
64         tokenCodes.Add("__DO", 3);
65         tokenCodes.Add("__IF", 4);
66         tokenCodes.Add("THEN", 5);
67         tokenCodes.Add("ELSE", 6);
68         tokenCodes.Add("_FOR", 7);
69         tokenCodes.Add("__OF", 8);
70         tokenCodes.Add("WTLN", 9);
71         tokenCodes.Add("RDLN", 10);
72         tokenCodes.Add("_BEG", 11);
73         tokenCodes.Add("_END", 12);
74         tokenCodes.Add("_VAR", 13);
75         tokenCodes.Add("WHIL", 14);
76         tokenCodes.Add("UNIT", 15);
77         tokenCodes.Add("LABL", 16);
78         tokenCodes.Add("REPT", 17);
79         tokenCodes.Add("UNTL", 18);
80         tokenCodes.Add("PROC", 19);
81         tokenCodes.Add("DOWN", 20);
82         tokenCodes.Add("FUNC", 21);
83         tokenCodes.Add("RTRN", 22);
84         tokenCodes.Add("REAL", 23);
85         tokenCodes.Add("_STR", 24);
86         tokenCodes.Add("ARRY", 25);
87
88         // Other Tokens
89         tokenCodes.Add("_DIV", 30);
90         tokenCodes.Add("_MUL", 31);
91         tokenCodes.Add("_ADD", 32);
92         tokenCodes.Add("_SUB", 33);
93         tokenCodes.Add("LPAR", 34);
94         tokenCodes.Add("RPAR", 35);
95         tokenCodes.Add("SEMI", 36);
96         tokenCodes.Add("ASGN", 37);
97         tokenCodes.Add("__GT", 38);
98         tokenCodes.Add("__LT", 39);
99         tokenCodes.Add("GTEQ", 40);
100        tokenCodes.Add("LTEQ", 41);
101        tokenCodes.Add("__EQ", 42);
```

```
102         tokenCodes.Add("NTEQ", 43);
103         tokenCodes.Add("COMM", 44);
104         tokenCodes.Add("LBRC", 45);
105         tokenCodes.Add("RBRC", 46);
106         tokenCodes.Add("COLN", 47);
107         tokenCodes.Add("_DOT", 48);
108
109         // Identifiers
110         tokenCodes.Add("IDNT", 50);
111
112         // Numeric Constants
113         tokenCodes.Add("INTC", 51);
114         tokenCodes.Add("FLTC", 52);
115
116         // String
117         tokenCodes.Add("STRC", 53);
118
119         // Used for any other input characters which are not defined.
120         tokenCodes.Add("UNDF", 99);
121
122         return tokenCodes;
123     }
124
125     /// <summary>
126     /// Initializes reserve table with reserve words and token codes
127     /// </summary>
128     /// <returns>Reserve table with reserve words and token codes</returns>
129     static ReserveTable InitializeReserveWordTable()
130     {
131         ReserveTable reserveWords = new ReserveTable();
132
133         // Token Codes
134         reserveWords.Add("GOTO", 0);
135         reserveWords.Add("INTEGER", 1);
136         reserveWords.Add("TO", 2);
137         reserveWords.Add("DO", 3);
138         reserveWords.Add("IF", 4);
139         reserveWords.Add("THEN", 5);
140         reserveWords.Add("ELSE", 6);
141         reserveWords.Add("FOR", 7);
142         reserveWords.Add("OF", 8);
143         reserveWords.Add("WRITELN", 9);
144         reserveWords.Add("READLN", 10);
145         reserveWords.Add("BEGIN", 11);
146         reserveWords.Add("END", 12);
147         reserveWords.Add("VAR", 13);
148         reserveWords.Add("WHILE", 14);
149         reserveWords.Add("UNIT", 15);
150         reserveWords.Add("LABEL", 16);
151         reserveWords.Add("REPEAT", 17);
152         reserveWords.Add("UNTIL", 18);
153         reserveWords.Add("PROCEDURE", 19);
154         reserveWords.Add("DOWNT", 20);
155     }
```

```
156     reserveWords.Add("RETURN", 22);
157     reserveWords.Add("REAL", 23);
158     reserveWords.Add("STRING", 24);
159     reserveWords.Add("ARRAY", 25);
160
161     // Other Tokens
162     reserveWords.Add("/", 30);
163     reserveWords.Add("*", 31);
164     reserveWords.Add("+", 32);
165     reserveWords.Add("-", 33);
166     reserveWords.Add("(", 34);
167     reserveWords.Add(")", 35);
168     reserveWords.Add(";", 36);
169     reserveWords.Add(":=", 37);
170     reserveWords.Add(">", 38);
171     reserveWords.Add("<", 39);
172     reserveWords.Add(">=", 40);
173     reserveWords.Add("<=", 41);
174     reserveWords.Add "=", 42);
175     reserveWords.Add("<>", 43);
176     reserveWords.Add(",", 44);
177     reserveWords.Add("[", 45);
178     reserveWords.Add("]", 46);
179     reserveWords.Add(":", 47);
180     reserveWords.Add(".", 48);
181
182     return reserveWords;
183 }
184
185 /// <summary>
186 /// Reads all the text from the source file and stores each line as a
187 /// separate element in a string array.
188 /// </summary>
189 /// <param name="filePath">Path to the file to be read into memory</
190 /// param>
191 /// <returns>The source text as a string array</returns>
192 static string[] InitializeInputFile(string filePath)
193 {
194     return File.ReadAllLines(filePath);
195 }
196
197 /// <summary>
198 /// Prints the Lexeme, the token code, a table-looked-up 4-character
199 /// mnemonic for that code,
200 /// and for identifiers and literals added to the symbol table, the
201 /// symbol table location index of the token.
202 /// </summary>
203 /// <param name="nextToken">The token most recently found</param>
204 /// <param name="tokenCode">The token code of the most recently found
205 /// token</param>
206 /// <param name="mnemonicTable">Table containing the mnemonic associated
207 /// with each token code</param>
208 /// <param name="symbolTable">Table containing identifiers, numeric
```

```
constants, and string constants</param>
203 static void PrintToken(string nextToken, int tokenCode, ReserveTable
mnemonicTable, SymbolTable symbolTable)
204 {
205     string mnemonic = mnemonicTable.LookupCode(tokenCode);
206     int symbolTableIndex;
207
208     if (tokenCode == 50)
209         symbolTableIndex = symbolTable.LookupSymbol(nextToken.ToUpper());
210     else
211         symbolTableIndex = symbolTable.LookupSymbol(nextToken);
212
213     if (symbolTableIndex == -1)
214     {
215         Console.WriteLine($"\\t|Token: {nextToken, -40} | Token Code:
{tokenCode, 2} | Mnemonic: {mnemonic, 4} | Symbol Table
Index:  |");
216     }
217     else
218     {
219         Console.WriteLine($"\\t|Token: {nextToken, -40} | Token Code:
{tokenCode, 2} | Mnemonic: {mnemonic, 4} | Symbol Table
Index: {symbolTableIndex, 2}|");
220     }
221 }
222 }
223 }
```

```
1 using System;
2 using System.Collections.Generic;
3 using System.ComponentModel.DataAnnotations;
4 using System.Data;
5 using System.Dynamic;
6 using System.IO;
7 using System.Linq;
8 using System.Runtime.InteropServices.ComTypes;
9 using System.Text;
10
11 namespace KyleBushCompiler
12 {
13     class LexicalAnalyzer
14     {
15         /// <summary>
16         /// Contains all possible states from the DFA diagram.
17         /// </summary>
18         enum State
19         {
20             START,
21             INTEGER_START,
22             INTEGER_ACCEPT,
23             FLOATING_POINT_START,
24             FLOATING_POINT_SCI_NOTATION,
25             FLOATING_POINT_SCI_NOTATION_SIGN,
26             FLOATING_POINT_SCI_NOTATION_DIGIT,
27             FLOATING_POINT_FRACTIONAL_DIGIT,
28             FLOATING_POINT_ACCEPT,
29             IDENTIFIER_START,
30             IDENTIFIER_ACCEPT,
31             STRING_START,
32             STRING_ACCEPT,
33             COMMENT_2_START,
34             COMMENT_2_BODY,
35             COMMENT_2_CLOSE,
36             COMMENT_1_BODY,
37             ONE_OR_TWO_CHAR_TOKEN_ACCEPT,
38             UNDEFINED
39         }
40
41         private State CurrentState;
42         private const int IDENTIFIER = 50;
43         private const int INTEGER = 51;
44         private const int FLOATING_POINT = 52;
45         private const int STRING = 53;
46         private const int UNDEFINED = 99;
47
48         private const int MAX_IDENTIFIER_LENGTH = 30;
49         private const int MAX_NUMERIC_LENGTH = 16;
50
51         public string NextToken { get; set; }
52         public int TokenCode { get; set; }
53         public SymbolTable SymbolTable { get; private set; }
```



```

54     public ReserveTable ReserveTable { get; private set; }
55     public bool EndOfFile { get; set; }
56     public string[] FileText { get; set; }
57     public string CurrentLine { get; set; }
58     public char CurrentChar { get; set; }
59     public char NextChar { get; set; }
60     public int CurrentLineIndex { get; set; }
61     public int CurrentCharIndex { get; set; }
62     public bool TokenFound { get; set; }
63     public bool EchoOn { get; set; }
64     public bool EndOfLine { get; private set; }
65
66     /// <summary>
67     /// Initializes the Lexical Analyzer to a baseline state.
68     /// </summary>
69     /// <param name="fileText">The source text as a string array</param>
70     /// <param name="symbolTable">The table that will hold all symbols      ↗
71     found</param>
72     /// <param name="reserveTable">The table containing the reserve words for ↗
73     the langauge</param>
74     public void Initialize(string[] fileText, SymbolTable symbolTable,
75     ReserveTable reserveTable)
76     {
77         SymbolTable = symbolTable;
78         ReserveTable = reserveTable;
79         EndOfFile = false;
80         EchoOn = false;
81         FileText = fileText;
82         CurrentLineIndex = 0;
83         CurrentCharIndex = 0;
84         CurrentLine = FileText[CurrentLineIndex];
85     }
86
87     /// <summary>
88     /// Identifies and returns the next available token in the source code.
89     /// </summary>
90     /// <param name="echoOn">Selects whether input lines are echoed when      ↗
91     read</param>
92     public void GetNextToken(bool echoOn)
93     {
94         CurrentState = State.START;
95         EchoOn = echoOn;
96         NextToken = "";
97         TokenFound = false;
98
99         while (!EndOfFile && !TokenFound)
100         {
101             GetNextChar();
102             // Check for single character comment identifier
103             if (CurrentChar == '{')
104             {
105                 CommentStyleOne();
106             }
107         }
108     }

```

```

104         else if (CurrentChar == '(' && LookAhead() == '*')
105         {
106             CommentStyleTwo();
107         }
108         // Check for one or two char tokens
109         else if (IsOneOrTwoCharTokenStart(CurrentChar))
110         {
111             GetOneOrTwoCharToken(CurrentChar);
112         }
113         // Check if NUMERIC CONSTANT either INTEGER or FLOATING_POINT
114         else if (IsDigit(CurrentChar))
115         {
116             GetNumericToken();
117         }
118         // Check if IDENTIFIER
119         else if (IsLetter(CurrentChar))
120         {
121             GetIdentifierToken();
122         }
123         // Check if STRING
124         else if (CurrentChar == '"')
125         {
126             GetStringToken();
127         }
128         // Found an undefined character
129         else
130         {
131             AddCharToNextToken();
132             AcceptToken(UNDEFINED, State.UNDEFINED);
133         }
134     }
135
136     if (EndOfFile)
137     {
138         CheckForEndOfFileErrors();
139     }
140 }
141
142 /// <summary>
143 /// Checks if the end of the file was reached before a comment or string
144 /// was closed.
145 /// </summary>
146 private void CheckForEndOfFileErrors()
147 {
148     switch (CurrentState)
149     {
150     case State.COMMENT_1_BODY:
151     case State.COMMENT_2_START:
152     case State.COMMENT_2_BODY:
153     case State.COMMENT_2_CLOSE:
154         Console.WriteLine("\tWARNING: End of file found before
155         comment terminated");
156         break;
157     case State.STRING_START:
158         Console.WriteLine("\tWARNING: Unterminated string found");
159     }

```

```
158     }
159 }
160
161 /// <summary>
162 /// A string token has been detected. This method will continue to add
163   characters to the
164   token until the end of the token or end of line is found.
165   </summary>
166 private void GetStringToken()
167 {
168     CurrentState = State.STRING_START;
169     NextChar = LookAhead();
170     while (!EndOfFile && NextChar != '"')
171     {
172         GetNextChar();
173         if (EndOfLine)
174         {
175             Console.WriteLine("\tWARNING: End of line was reached before
176   \" was found to close string.");
177             break;
178         }
179         AddCharToNextToken();
180         NextChar = LookAhead();
181     }
182     AcceptToken(String, State.STRING_ACCEPT);
183     AddTokenToSymbolTable();
184     if (NextChar == '"')
185         GetNextChar();
186 }
187
188 /// <summary>
189 /// An identifier has been detected. This method will continue to add
190   characters to the token
191   until the end of the token is found.
192   </summary>
193 private void GetIdentifierToken()
194 {
195     CurrentState = State.IDENTIFIER_START;
196     AddCharToNextToken();
197     while (!EndOfFile && !IsWhitespace(LookAhead()) && IsLetter(LookAhead()
198   ()) || IsDigit(LookAhead()) || LookAhead() == '_' || LookAhead() ==
199   '$')
200     {
201         GetNextChar();
202         AddCharToNextToken();
203     }
204     AcceptToken(GetIdentifierCode(), State.IDENTIFIER_ACCEPT);
205     if (TokenCode == IDENTIFIER)
206         AddTokenToSymbolTable();
207 }
208
209 /// <summary>
210 /// A numeric token has been detected. This determines if the token is an
```

```
208     /// integer or floating point token and builds that token.
209     /// </summary>
210     private void GetNumericToken()
211     {
212         CurrentState = State.INTEGER_START;
213         AddCharToNextToken();
214
215         NextChar = LookAhead();
216
217         while (!EndOfFile && IsDigit(NextChar))
218         {
219             GetNextChar();
220             AddCharToNextToken();
221             NextChar = LookAhead();
222             if (EndOfLine)
223                 break;
224         }
225         if (NextChar == '.')
226         {
227             GenerateFloatingPointToken();
228         }
229         else
230         {
231             AcceptToken(INTEGER, State.INTEGER_ACCEPT);
232             AddTokenToSymbolTable();
233         }
234     }
235
236     /// <summary>
237     /// A floating point token has been detected. This method will build that token.
238     /// </summary>
239     private void GenerateFloatingPointToken()
240     {
241         CurrentState = State.FLOATING_POINT_START;
242         GetNextChar();
243         AddCharToNextToken();
244
245         NextChar = LookAhead();
246         if (IsDigit(NextChar))
247         {
248             while (!EndOfFile && IsDigit(NextChar))
249             {
250                 GetNextChar();
251                 AddCharToNextToken();
252                 NextChar = LookAhead();
253                 if (EndOfLine)
254                     break;
255             }
256             if (NextChar == 'E')
257             {
258                 GenerateFloatingPointScientificNotationToken();
259             }
260         }
```

```
261         else if (NextChar == 'E')
262         {
263             GenerateFloatingPointScientificNotationToken();
264         }
265
266         AcceptToken(FLOATING_POINT, State.FLOATING_POINT_ACCEPT);
267         AddTokenToSymbolTable();
268     }
269
270     /// <summary>
271     /// A floating point token using scientific notation has been detected.
272     /// This method builds that token.
273     /// </summary>
274     private void GenerateFloatingPointScientificNotationToken()
275     {
276         CurrentState = State.FLOATING_POINT_SCI_NOTATION;
277         GetNextChar();
278         AddCharToNextToken();
279         NextChar = LookAhead();
280
281         if (NextChar == '-' || NextChar == '+')
282         {
283             CurrentState = State.FLOATING_POINT_SCI_NOTATION_SIGN;
284             GetNextChar();
285             AddCharToNextToken();
286             NextChar = LookAhead();
287         }
288
289         if (IsDigit(NextChar))
290         {
291             CurrentState = State.FLOATING_POINT_SCI_NOTATION_DIGIT;
292             GetNextChar();
293             AddCharToNextToken();
294             NextChar = LookAhead();
295
296             while (!EndOfFile && IsDigit(NextChar))
297             {
298                 GetNextChar();
299                 AddCharToNextToken();
300                 NextChar = LookAhead();
301                 if (EndOfLine)
302                     break;
303             }
304             AcceptToken(FLOATING_POINT, State.FLOATING_POINT_ACCEPT);
305             AddTokenToSymbolTable();
306         }
307         else
308         {
309             Console.WriteLine("ERROR: Expected at least one digit.");
310         }
311     }
312
313     /// <summary>
314     /// Flags that a token has been found, sets the current state of the DFA,
315     /// sets the correct token code, and truncates the token if needed.
```

```
316     /// </summary>
317     /// <param name="tokenCode">The token code of the token that was found</param>
318     /// <param name="state">The current state of the DFA</param>
319     private void AcceptToken(int tokenCode, State state)
320     {
321         TokenFound = true;
322         CurrentState = state;
323         TokenCode = tokenCode;
324         TruncateTokenIfTooLong();
325     }
326
327     /// <summary>
328     /// A comment has been detected using the delimiter (*.
329     /// This method ignores all characters until a closing delimiter
330     /// or the end of the file is found.
331     /// </summary>
332     private void CommentStyleTwo()
333     {
334         CurrentState = State.COMMENT_2_BODY;
335         GetNextChar();
336         GetNextChar();
337         NextChar = LookAhead();
338
339         // TODO: This still exits too early because seeing * causes exit even
340         // if NextChar is not )
341         while (!EndOfFile && (CurrentChar != '*' && NextChar != ')') ||
342             (CurrentChar == '*' && NextChar != ')') || (CurrentChar != '*' &&
343             NextChar == ')'))
344         {
345             GetNextChar();
346             NextChar = LookAhead();
347         };
348
349         GetNextChar();
350
351         if (!EndOfFile)
352             CurrentState = State.START;
353     }
354
355     /// <summary>
356     /// A comment has been detected using the { delimiter.
357     /// This method ignores all characters until a closing delimiter
358     /// or the end of the file is found.
359     /// </summary>
360     private void CommentStyleOne()
361     {
362         CurrentState = State.COMMENT_1_BODY;
363         while (CurrentChar != '}')
364         {
365             GetNextChar();
366             if (EndOfFile)
```

```

367         if (!EndOfFile)
368             CurrentState = State.START;
369     }
370
371     /// <summary>
372     /// Truncates the token if it is too long for the defined token type
373     /// and displays a warning message.
374     /// </summary>
375     private void TruncateTokenIfTooLong()
376     {
377         // TODO: differentiate between numeric and identifiers.
378         int maxLength;
379
380         if (TokenCode == IDENTIFIER)
381             maxLength = MAX_IDENTIFIER_LENGTH;
382         else if (TokenCode == FLOATING_POINT || TokenCode == INTEGER)
383             maxLength = MAX_NUMERIC_LENGTH;
384         else
385             return;
386
387         if (NextToken.Length > maxLength)
388         {
389             Console.WriteLine("\tWARNING: Token length exceeds " + maxLength +
390                             " + ". Token has been truncated.");
391             NextToken = NextToken.Substring(0, maxLength);
392         }
393     }
394
395     /// <summary>
396     /// Determines if a token is one of the predefined one or two character
397     /// tokens
398     /// from section 6 of the CS4100projectlangFA20-TOKENS.pdf
399     /// </summary>
400     /// <param name="c">The character being tested.</param>
401     /// <returns>True if character is one or two char token. False if not.</
402     returns>
403     private bool IsOneOrTwoCharTokenStart(char c)
404     {
405         switch(c)
406         {
407             case '/':
408             case '*':
409             case '+':
410             case '-':
411             case '(':
412             case ')':
413             case ';':
414             case '=':
415             case ',':
416             case '[':
417             case ']':
418             case '.':
419             case ':':
420             case '>':

```

```
418         case '<':
419             return true;
420         default:
421             return false;
422     }
423 }
424
425 /// <summary>
426 /// One of the predefined one or two character tokens
427 /// from section 6 of the CS4100projectlangFA20-TOKENS.pdf
428 /// has been detected so this method stores it in NextToken.
429 /// </summary>
430 /// <param name="c">The current character</param>
431 private void GetOneOrTwoCharToken(char c)
432 {
433     CurrentState = State.ONE_OR_TWO_CHAR_TOKEN_ACCEPT;
434     switch (c)
435     {
436         case '/':
437         case '*':
438         case '+':
439         case '-':
440         case '(':
441         case ')':
442         case ';':
443         case '=':
444         case ',':
445         case '[':
446         case ']':
447         case '.':
448             NextToken += CurrentChar;
449             break;
450         case ':':
451             if (LookAhead() == '=')
452             {
453                 NextToken += CurrentChar;
454                 GetNextChar();
455                 NextToken += CurrentChar;
456             }
457             else
458             {
459                 NextToken += CurrentChar;
460             }
461             break;
462         case '>':
463             if (LookAhead() == '=')
464             {
465                 NextToken += CurrentChar;
466                 GetNextChar();
467                 NextToken += CurrentChar;
468             }
469             else
470             {
471                 NextToken += CurrentChar;
```



```

472     }
473     break;
474     case '<':
475         if (LookAhead() == '=' || LookAhead() == '>')
476         {
477             NextToken += CurrentChar;
478             GetNextChar();
479             NextToken += CurrentChar;
480         }
481         else
482         {
483             NextToken += CurrentChar;
484         }
485         break;
486     }
487     AcceptToken(ReserveTable.LookupName(NextToken),
488                State.ONE_OR_TWO_CHAR_TOKEN_ACCEPT);
489
490     /// <summary>
491     /// Peeks at the next character without advancing.
492     /// </summary>
493     /// <returns>The next character without advancing.</returns>
494     private char LookAhead()
495     {
496         char lookAhead = ' ';
497         if (CurrentCharIndex < CurrentLine.Length)
498         {
499             lookAhead = CurrentLine[CurrentCharIndex];
500         }
501         return lookAhead;
502     }
503
504     /// <summary>
505     /// Checks if the token is already in the symbol table.
506     /// If it is not then it is added, otherwise it does nothing.
507     /// </summary>
508     private void AddTokenToSymbolTable()
509     {
510         string tokenToAdd;
511         if (TokenCode == IDENTIFIER)
512             tokenToAdd = NextToken.ToUpper();
513         else
514             tokenToAdd = NextToken;
515
516         int symbolIndex = SymbolTable.LookupSymbol(tokenToAdd);
517         if (symbolIndex == -1)
518         {
519             switch (TokenCode)
520             {
521                 case IDENTIFIER:
522                     SymbolTable.AddSymbol(tokenToAdd, SymbolKind.Variable,
523                                           0);
524                     break;
525                 case INTEGER:

```

```

...sign\KyleBushCompiler\KyleBushCompiler\LexicalAnalyzer.cs 11
525         SymbolTable.AddSymbol(tokenToAdd, SymbolKind.Constant,
Int64.Parse(tokenToAdd));
526         break;
527     case FLOATING_POINT:
528         SymbolTable.AddSymbol(tokenToAdd, SymbolKind.Constant,
Double.Parse(tokenToAdd));
529         break;
530     case STRING:
531         SymbolTable.AddSymbol(tokenToAdd, SymbolKind.Constant,
tokenToAdd);
532         break;
533     }
534 }
535 }
536
537 /// <summary>
538 /// Queries the Reserve Table to determine if the current token is a
reserve word.
539 /// If it is then the proper token code is returned from the table.
540 /// If it is not a reserve word it is given the identifier token code.
541 /// </summary>
542 /// <returns></returns>
543 private int GetIdentifierCode()
544 {
545     int code = ReserveTable.LookupName(NextToken.ToUpper());
546     if (code == -1)
547     {
548         return IDENTIFIER;
549     }
550
551     return code;
552 }
553
554 /// <summary>
555 /// Adds the current char to NextToken.
556 /// </summary>
557 private void AddCharToNextToken()
558 {
559     NextToken += CurrentChar;
560 }
561
562 /// <summary>
563 /// Get's the next line of source text and prints it if EchoOn is true
564 /// </summary>
565 private void GetNextLine()
566 {
567     if (CurrentLineIndex < FileText.Length)
568     {
569         CurrentLine = FileText[CurrentLineIndex];
570         CurrentLineIndex++;
571     }
572
573     if (EchoOn)
574     {
575         Console.WriteLine(CurrentLine);

```

```

576     }
577 }
578
579 /// <summary>
580 /// Get's the next character from the source text.
581 /// Also, checks for the end of the file and the end of a line.
582 /// Skips blanks that are not part of a token.
583 /// </summary>
584 private void GetNextChar()
585 {
586     if (IsEndOfFile())
587     {
588         EndOfFile = true;
589         return;
590     }
591
592     if (IsEndOfLine())
593     {
594         if (IsCommentOrStart())
595         {
596             GetNextLine();
597             CurrentCharIndex = 0;
598             EndOfLine = false;
599         }
600         else
601         {
602             EndOfLine = true;
603             return;
604         }
605     }
606
607     if (!string.IsNullOrEmpty(CurrentLine))
608     {
609         CurrentChar = CurrentLine[CurrentCharIndex];
610         CurrentCharIndex++;
611     }
612
613     if (CurrentState == State.START)
614     {
615         SkipBlanks();
616     }
617 }
618
619 /// <summary>
620 /// Determines if the current state of the DFA is a comment or start.
621 /// </summary>
622 /// <returns>True if the DFA is in a comment of start state. False if not.</returns>
623 private bool IsCommentOrStart()
624 {
625     switch (CurrentState)
626     {
627         case State.START:
628         case State.COMMENT_1_BODY:
629         case State.COMMENT_2_START:
630         case State.COMMENT_2_BODY:

```

```
631         case State.COMMENT_2_CLOSE:
632             return true;
633     }
634     return false;
635 }
636
637 /// <summary>
638 /// Skips blanks and empty lines that are not part of tokens.
639 /// </summary>
640 private void SkipBlanks()
641 {
642     while (!EndOfFile && IsWhitespace(CurrentChar) || string.IsNullOrEmpty(CurrentLine)) ↗
643     {
644         GetNextChar();
645     }
646 }
647
648 /// <summary>
649 /// Checks if the end of the file has been found.
650 /// </summary>
651 /// <returns>True if end of line is found. False if not.</returns>
652 private bool IsEndOfFile()
653 {
654     return (CurrentLineIndex == FileText.Length && CurrentCharIndex == ↗
655         CurrentLine.Length);
656 }
657
658 /// <summary>
659 /// Checks if the end of a line has been found.
660 /// </summary>
661 /// <returns>True if end of line is found. False if not</returns>
662 private bool IsEndOfLine()
663 {
664     return CurrentCharIndex == CurrentLine.Length;
665 }
666
667 /// <summary>
668 /// Checks if a character is a letter.
669 /// </summary>
670 /// <param name="c"></param>
671 /// <returns>True if char is letter. False if not.</returns>
672 private bool IsLetter(char c)
673 {
674     return (c >= 'A' && c <= 'Z') || (c >= 'a' && c <= 'z');
675 }
676
677 /// <summary>
678 /// Checks if a character is a digit.
679 /// </summary>
680 /// <param name="c"></param>
681 /// <returns>True if char is digit. False if not.</returns>
682 private bool IsDigit(char c)
683 {
684     return (c >= '0' && c <= '9');
```

```
684     }
685
686     /// <summary>
687     /// Checks if a character is whitespace.
688     /// </summary>
689     /// <param name="c"></param>
690     /// <returns>True if char is whitespace. False if not.</returns>
691     private bool IsWhitespace(char c)
692     {
693         return char.IsWhiteSpace(c);
694     }
695 }
696 }
697
```

```
1 using System;
2 using System.Collections.Generic;
3 using System.Linq;
4 using System.Security.Cryptography.X509Certificates;
5 using System.Text;
6
7 namespace KyleBushCompiler
8 {
9     /// <summary>
10    /// Contains all the reserve words for a language.
11    /// </summary>
12    public class ReserveTable
13    {
14        private const int TABLEWIDTH = 16;
15        private const char DIVIDER_CHAR = '-';
16        public List<ReservedWord> ReserveTableData { get; set; }
17
18        /// <summary>
19        /// Creates a new ReserveTable and initializes a list of ReservedWords.
20        /// </summary>
21        public ReserveTable()
22        {
23            ReserveTableData = new List<ReservedWord>();
24        }
25
26        /// <summary>
27        /// Initializes the table with all the reserve words for the language.
28        /// </summary>
29        public void Initialize(List<ReservedWord> reservedWords)
30        {
31            ReserveTableData = reservedWords;
32        }
33
34        /// <summary>
35        /// Returns the index of the row where the data was place, just adds to
36        /// end of list.
37        /// </summary>
38        /// <param name="name">String name of reserved word</param>
39        /// <param name="code">Integer code of reserved word</param>
40        /// <returns>index of the row where the data was placed</returns>
41        public int Add(string name, int code)
42        {
43            ReservedWord reservedWord = new ReservedWord(name, code);
44            ReserveTableData.Add(reservedWord);
45            return ReserveTableData.Count - 1;
46        }
47
48        /// <summary>
49        /// Returns the code associated with name if name is in the table, else
50        /// returns -1
51        /// </summary>
52        /// <param name="name">String name of reserved word</param>
53        /// <returns></returns>
54        public int LookupName(string name)
```

```

53     {
54         ReservedWord reservedWord = ReserveTableData.FirstOrDefault(x =>
55             x.Name == name);
56         if (reservedWord == null)
57         {
58             return -1;
59         }
60         return reservedWord.Code;
61     }
62     /// <summary>
63     /// Returns the associated name if code is there, else an empty string
64     /// </summary>
65     /// <param name="code">Integer code of reserved word</param>
66     /// <returns></returns>
67     public string LookupCode(int code)
68     {
69         ReservedWord reservedWord = ReserveTableData.FirstOrDefault(x =>
70             x.Code == code);
71         if (reservedWord == null)
72         {
73             return "";
74         }
75         return reservedWord.Name;
76     }
77     /// <summary>
78     /// Searches the table for the given code to test if it is valid.
79     /// </summary>
80     /// <param name="code">Integer code of reserved word</param>
81     /// <returns>True if the code is valid, False if not.</returns>
82     public bool isValidOpCode(int code)
83     {
84         ReservedWord reservedWord = ReserveTableData.FirstOrDefault(x =>
85             x.Code == code);
86         if (reservedWord == null)
87         {
88             Console.WriteLine($"{code} is not a valid Op Code.");
89             return false;
90         }
91         return true;
92     }
93     /// <summary>
94     /// Prints the currently used contents of the Reserve table in neat
95     /// tabular format
96     /// </summary>
97     public void PrintReserveTable()
98     {
99         Console.WriteLine("RESERVE TABLE");
100         DrawHorizontalBorder(TABLEWIDTH, DIVIDER_CHAR);
101         Console.WriteLine($"|{ "Name", -7 }|{ "Code", 5 }|");
102         DrawHorizontalBorder(TABLEWIDTH, DIVIDER_CHAR);
103         foreach (var code in ReserveTableData)

```

```
104         Console.WriteLine($"{ code.Name, -7 }|{ code.Code, 5 }|");
105     }
106     DrawHorizontalBorder(TABLEWIDTH, DIVIDER_CHAR);
107 }
108
109     /// <summary>
110     /// Draws a horizontal border using the given character repeated by the
111     given length
112     /// </summary>
113     /// <param name="length">number of times to repeat character</param>
114     /// <param name="character">character used to draw the border</param>
115     public void DrawHorizontalBorder(int length, char character)
116     {
117         for (int i = 0; i < length; i++)
118         {
119             Console.Write(character);
120         }
121         Console.WriteLine();
122     }
123 }
124 }
125
```



```
1 using System;
2 using System.Collections.Generic;
3 using System.Text;
4
5 namespace KyleBushCompiler
6 {
7     /// <summary>
8     /// Contains the string and integer representations of an OpCode.
9     /// </summary>
10    public class ReservedWord
11    {
12        public string Name { get; set; }
13        public int Code { get; set; }
14
15        /// <summary>
16        /// Creates a new ReservedWord object.
17        /// </summary>
18        /// <param name="name">String name of reserved word</param>
19        /// <param name="code">Integer code of reserved word</param>
20        public ReservedWord(string name, int code)
21        {
22            Name = name;
23            Code = code;
24        }
25    }
26 }
27
```

```
1 using System;
2 using System.Collections.Generic;
3 using System.Dynamic;
4 using System.Text;
5
6 namespace KyleBushCompiler
7 {
8     /// <summary>
9     /// Used to specify data type of a symbol
10    /// </summary>
11    public enum DataType
12    {
13        Integer,
14        Double,
15        String
16    }
17
18    /// <summary>
19    /// Used to specify the kind of a symbol
20    /// </summary>
21    public enum SymbolKind
22    {
23        Label,
24        Variable,
25        Constant
26    }
27
28    public class Symbol
29    {
30        public string Name { get; set; }
31        public SymbolKind Kind { get; set; }
32        public DataType DataType { get; set; }
33
34        private int _intValue;
35        private string _stringValue;
36        private double _doubleValue;
37
38        /// <summary>
39        /// Constructor to initialize a Symbol containing an integer value.
40        /// </summary>
41        /// <param name="name">String name of symbol</param>
42        /// <param name="kind">Defines the kind of the symbol</param>
43        /// <param name="dataType">Defines the data type of the symbol</param>
44        /// <param name="value">The integer value of the symbol</param>
45        public Symbol(string name, SymbolKind kind, DataType dataType, int value)
46        {
47            Name = name;
48            Kind = kind;
49            DataType = dataType;
50            _intValue = value;
51        }
52
53        /// <summary>
54        /// Constructor to initialize a Symbol containing a double value.
```

```
55     /// </summary>
56     /// <param name="name">String name of symbol</param>
57     /// <param name="kind">Defines the kind of the symbol</param>
58     /// <param name="dataType">Defines the data type of the symbol</param>
59     /// <param name="value">The double value of the symbol</param>
60     public Symbol(string name, SymbolKind kind, DataType dataType, double value)
61     {
62         Name = name;
63         Kind = kind;
64         DataType = dataType;
65         _doubleValue = value;
66     }
67
68     /// <summary>
69     /// Contructor to initialize a Symbol containing a string value.
70     /// </summary>
71     /// <param name="name">String name of symbol</param>
72     /// <param name="kind">Defines the kind of the symbol</param>
73     /// <param name="dataType">Defines the data type of the symbol</param>
74     /// <param name="value">The string value of the symbol</param>
75     public Symbol(string name, SymbolKind kind, DataType dataType, string value)
76     {
77         Name = name;
78         Kind = kind;
79         DataType = dataType;
80         _stringValue = value;
81     }
82
83     /// <summary>
84     /// Sets a Symbol with an integer value.
85     /// </summary>
86     /// <param name="value">The integer value of the symbol</param>
87     public void SetValue(int value)
88     {
89         _intValue = value;
90     }
91
92     /// <summary>
93     /// Sets a Symbol with a string value.
94     /// </summary>
95     /// <param name="value">The string value of the symbol</param>
96     public void SetValue(string value)
97     {
98         _stringValue = value;
99     }
100
101     /// <summary>
102     /// Sets a Symbol with a double value.
103     /// </summary>
104     /// <param name="value">The double value of the symbol</param>
105     public void SetValue(double value)
106     {
```

```
107         _doubleValue = value;
108     }
109
110     /// <summary>
111     /// Checks the DataType of the Symbol and returns the appropriate value.
112     /// </summary>
113     /// <returns>int, string, or double depending on the DataType property.</ returns>
114     public dynamic GetValue()
115     {
116         if (DataType == DataType.Integer)
117             return _intValue;
118         else if (DataType == DataType.Double)
119             return _doubleValue;
120         else
121             return _stringValue;
122     }
123 }
124 }
125
```

```
1 using System;
2 using System.Collections.Generic;
3 using System.Linq;
4 using System.Security.Cryptography.X509Certificates;
5 using System.Text;
6
7 namespace KyleBushCompiler
8 {
9     /// <summary>
10    /// Contains all the symbols for a given application.
11    /// </summary>
12    public class SymbolTable
13    {
14        private const int TABLEWIDTH = 105;
15        private const char DIVIDER_CHAR = '-';
16        private List<Symbol> SymbolTableData { get; set; }
17
18        /// <summary>
19        /// Creates a new, empty Symbol Table.
20        /// </summary>
21        public SymbolTable()
22        {
23            SymbolTableData = new List<Symbol>();
24        }
25
26        /// <summary>
27        /// Adds symbol with given kind and value to the symbol table,
28        /// automatically setting the correct data_type,
29        /// and returns the index where the symbol was located. If the symbol is
30        /// already in the table,
31        /// no change or verification is made, and this just returns the index
32        /// where the symbol was found.
33        /// </summary>
34        /// <param name="symbol">The symbol to add to the symbol table</param>
35        /// <param name="kind">The kind of symbol</param>
36        /// <param name="value">The value associated with the given symbol</
37        param>
38        /// <returns>The index of the added symbol in the symbol table as an
39        integer</returns>
40        public int AddSymbol(string symbol, SymbolKind kind, int value)
41        {
42            SymbolTableData.Add(new Symbol(symbol, kind, DataType.Integer,
43            value));
44            return SymbolTableData.Count - 1;
45        }
46
47        /// <summary>
48        /// Adds symbol with given kind and value to the symbol table,
49        /// automatically setting the correct data_type,
50        /// and returns the index where the symbol was located. If the symbol is
51        /// already in the table,
52        /// no change or verification is made, and this just returns the index
53        /// where the symbol was found.
54        /// </summary>
```

```
46     /// <param name="symbol">The symbol to add to the symbol table</param>
47     /// <param name="kind">The kind of symbol</param>
48     /// <param name="value">The value associated with the given symbol</param>
49     /// <returns>The index of the added symbol in the symbol table as an integer</returns>
50     public int AddSymbol(string symbol, SymbolKind kind, double value)
51     {
52         SymbolTableData.Add(new Symbol(symbol, kind, DataType.Double, value));
53         return SymbolTableData.Count - 1;
54     }
55
56     /// <summary>
57     /// Adds symbol with given kind and value to the symbol table, automatically setting the correct data_type,
58     /// and returns the index where the symbol was located. If the symbol is already in the table,
59     /// no change or verification is made, and this just returns the index where the symbol was found.
60     /// </summary>
61     /// <param name="symbol">The symbol to add to the symbol table</param>
62     /// <param name="kind">The kind of symbol</param>
63     /// <param name="value">The value associated with the given symbol</param>
64     /// <returns>The index of the added symbol in the symbol table as an integer</returns>
65     public int AddSymbol(string symbol, SymbolKind kind, string value)
66     {
67         SymbolTableData.Add(new Symbol(symbol, kind, DataType.String, value));
68         return SymbolTableData.Count - 1;
69     }
70
71     /// <summary>
72     /// Returns the index where symbol is found, or -1 if not in the table
73     /// </summary>
74     /// <param name="symbol">The symbol to look for in the table.</param>
75     /// <returns>The index of the symbol or -1 if not found</returns>
76     public int LookupSymbol(string symbol)
77     {
78         return SymbolTableData.FindIndex(s => s.Name == symbol);
79     }
80
81     /// <summary>
82     /// Return kind, data type, and value fields stored at index
83     /// </summary>
84     /// <param name="index">The index of the symbol to return</param>
85     /// <returns></returns>
86     public Symbol GetSymbol(int index)
87     {
88         return SymbolTableData[index];
89     }
90
```

```

91     /// <summary>
92     /// Set appropriate fields at slot indicated by index
93     /// </summary>
94     /// <param name="index">The index of the symbol to update</param>
95     /// <param name="kind">The kind of symbol</param>
96     /// <param name="value">The value of the symbol</param>
97     public void UpdateSymbol(int index, SymbolKind kind, int value)
98     {
99         SymbolTableData[index].Kind = kind;
100         SymbolTableData[index].SetValue(value);
101     }
102
103     /// <summary>
104     /// Set appropriate fields at slot indicated by index
105     /// </summary>
106     /// <param name="index">The index of the symbol to update</param>
107     /// <param name="kind">The kind of symbol</param>
108     /// <param name="value">The value of the symbol</param>
109     public void UpdateSymbol(int index, SymbolKind kind, double value)
110     {
111         SymbolTableData[index].Kind = kind;
112         SymbolTableData[index].SetValue(value);
113     }
114
115     /// <summary>
116     /// Set appropriate fields at slot indicated by index
117     /// </summary>
118     /// <param name="index">The index of the symbol to update</param>
119     /// <param name="kind">The kind of symbol</param>
120     /// <param name="value">The value of the symbol</param>
121     public void UpdateSymbol(int index, SymbolKind kind, string value)
122     {
123         SymbolTableData[index].Kind = kind;
124         SymbolTableData[index].SetValue(value);
125     }
126
127     /// <summary>
128     /// Prints the utilized rows of the symbol table in neat tabular format,
129     /// showing only the value field which is active for that row
130     /// </summary>
131     public void PrintSymbolTable()
132     {
133         Console.WriteLine("SYMBOL TABLE");
134         DrawHorizontalBorder(TABLEWIDTH, DIVIDER_CHAR);
135         Console.WriteLine($"{ "Name",-40 }|{ "Kind",10 }|{ "DataType",10 }|  ↗
136             { "Value",40 }|");
137         DrawHorizontalBorder(TABLEWIDTH, DIVIDER_CHAR);
138         foreach (var symbol in SymbolTableData)
139         {
140             Console.WriteLine($"{ symbol.Name,-40 }|{ symbol.Kind,10 }|  ↗
141                 { symbol.DataType,10 }|{ symbol.GetValue(),40 }|");
142         }
143         DrawHorizontalBorder(TABLEWIDTH, DIVIDER_CHAR);

```

```
143     }
144
145     /// <summary>
146     /// Draws a horizontal border using the given character repeated by the given length
147     /// </summary>
148     /// <param name="length">number of times to repeat character</param>
149     /// <param name="character">character used to draw the border</param>
150     public void DrawHorizontalBorder(int length, char character)
151     {
152         for (int i = 0; i < length; i++)
153         {
154             Console.Write(character);
155         }
156         Console.WriteLine();
157     }
158 }
159 }
160
```