```
 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
              found</param>
71          /// <param name="reserveTable">The table containing the reserve words for
               the langauge</param>
72          public void Initialize(string[] fileText, SymbolTable symbolTable,
              ReserveTable reserveTable)
73          {
74              SymbolTable = symbolTable;
75              ReserveTable = reserveTable;
76              EndOfFile = false;
77              EchoOn = false;
78              FileText = fileText;
79              CurrentLineIndex = 0;
80              CurrentCharIndex = 0;
81              CurrentLine = FileText[CurrentLineIndex];
82          }
83
84          /// <summary>
85          /// Identifies and returns the next available token in the source code.
86          /// </summary>
87          /// <param name="echoOn">Selects whether input lines are echoed when
              read</param>
88          public void GetNextToken(bool echoOn)
89          {
90              CurrentState = State.START;
91              EchoOn = echoOn;
92              NextToken = "";
93              TokenFound = false;
94
95              while (!EndOfFile && !TokenFound)
96              {
97                  GetNextChar();
98                  // Check for single character comment identifier
99                  if (CurrentChar == '{')
100                 {
101                     CommentStyleOne();
102                 }
103
```

```
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
              was closed.
144         /// </summary>
145         private void CheckForEndOfFileErrors()
146         {
147             switch (CurrentState)
148             {
149                 case State.COMMENT_1_BODY:
150                 case State.COMMENT_2_START:
151                 case State.COMMENT_2_BODY:
152                 case State.COMMENT_2_CLOSE:
153                     Console.WriteLine("\tWARNING: End of file found before
                          comment terminated");
154                     break;
155                 case State.STRING_START:
156                     Console.WriteLine("\tWARNING: Unterminated string found");
157 
```

```
158             }
159         }
160
161         /// <summary>
162         /// A string token has been detected. This method will continue to add
              characters to the
163         /// token until the end of the token or end of line is found.
164         /// </summary>
165         private void GetStringToken()
166         {
167             CurrentState = State.STRING_START;
168             NextChar = LookAhead();
169             while (!EndOfFile && NextChar != '"')
170             {
171                 GetNextChar();
172                 if (EndOfLine)
173                 {
174                     Console.WriteLine("\tWARNING: End of line was reached before
                       \" was found to close string.");
175                     break;
176                 }
177                 AddCharToNextToken();
178                 NextChar = LookAhead();
179             }
180
181             AcceptToken(STRING, State.STRING_ACCEPT);
182             AddTokenToSymbolTable();
183
184             if (NextChar == '"')
185                 GetNextChar();
186         }
187
188         /// <summary>
189         /// An identifier has been detected. This method will continue to add
              characters to the token
190         /// until the end of the token is found.
191         /// </summary>
192         private void GetIdentifierToken()
193         {
194             CurrentState = State.IDENTIFIER_START;
195             AddCharToNextToken();
196             while (!EndOfFile && !IsWhitespace(LookAhead()) && IsLetter(LookAhead
                  ()) || IsDigit(LookAhead()) || LookAhead() == '_' || LookAhead() ==
                   '$')
197             {
198                 GetNextChar();
199                 AddCharToNextToken();
200             }
201             AcceptToken(GetIdentifierCode(), State.IDENTIFIER_ACCEPT);
202             if (TokenCode == IDENTIFIER)
203                 AddTokenToSymbolTable();
204         }
205
206         /// <summary>
207         /// 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
                   if NextChar is not )
340             while (!EndOfFile && (CurrentChar != '*' && NextChar != ')') ||
                  (CurrentChar == '*' && NextChar != ')') || (CurrentChar != '*' &&
                  NextChar == ')'))
341             {
342                 GetNextChar();
343                 NextChar = LookAhead();
344             };
345 
346             GetNextChar();
347 
348             if (!EndOfFile)
349                 CurrentState = State.START;
350         }
351 
352         /// <summary>
353         /// A comment has been detected using the { delimiter.
354         /// This method ignores all characters until a closing delimiter
355         /// or the end of the file is found.
356         /// </summary>
357         private void CommentStyleOne()
358         {
359             CurrentState = State.COMMENT_1_BODY;
360             while (CurrentChar != '}')
361             {
362                 GetNextChar();
363                 if (EndOfFile)
364                     return;
365             };
366 
```

```
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
                      + ". Token has been truncated.");
390                 NextToken = NextToken.Substring(0, maxLength);
391             }
392         }
393
394         /// <summary>
395         /// Determines if a token is one of the predefined one or two character
              tokens
396         /// from section 6 of the CS4100projectlangFA20-TOKENS.pdf
397         /// </summary>
398         /// <param name="c">The character being tested.</param>
399         /// <returns>True if character is one or two char token. False if not.</
              returns>
400         private bool IsOneOrTwoCharTokenStart(char c)
401         {
402             switch(c)
403             {
404                 case '/':
405                 case '*':
406                 case '+':
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                     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),
                  State.ONE_OR_TWO_CHAR_TOKEN_ACCEPT);
488         }
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,
                          0);
523                         break;
524                     case INTEGER:
```

```
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 ==
                  CurrentLine.Length);
655         }
656 
657         /// <summary>
658         /// Checks if the end of a line has been found.
659         /// </summary>
660         /// <returns>True if end of line is found. False if not</returns>
661         private bool IsEndOfLine()
662         {
663             return CurrentCharIndex == CurrentLine.Length;
664         }
665 
666         /// <summary>
667         /// Checks if a character is a letter.
668         /// </summary>
669         /// <param name="c"></param>
670         /// <returns>True if char is letter. False if not.</returns>
671         private bool IsLetter(char c)
672         {
673             return (c >= 'A' && c <= 'Z') || (c >= 'a' && c <= 'z');
674         }
675 
676         /// <summary>
677         /// Checks if a character is a digit.
678         /// </summary>
679         /// <param name="c"></param>
680         /// <returns>True if char is digit. False if not.</returns>
681         private bool IsDigit(char c)
682         {
683             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
```