

```
1 using System;
2 using System.Collections.Generic;
3 using System.ComponentModel.DataAnnotations;
4 using System.ComponentModel.Design;
5 using System.IO;
6 using System.Reflection.Emit;
7
8 namespace KyleBushCompiler
9 {
10     class Program
11     {
12         /*
13          * CFG for Language Definition
14          * <program> -> $UNIT <prog-identifier> $SEMICOLON <block> $PERIOD
15          * <block> -> $BEGIN <statement> {$SEMICOLON <statement>}* $END
16          * <prog-identifier> -> <identifier>
17          * <statement> -> <variable> $COLON_EQUALS <simple expression>
18          * <variable> -> <identifier>
19          * <simple expression> -> [<sign>] <term> {<addop> <term>}*
20          * <addop> -> $PLUS | $MINUS
21          * <sign> -> $PLUS | $MINUS
22          * <term> -> <factor> {<mulop> <factor> }*
23          * <mulop> -> $MULTIPLY | $DIVIDE
24          * <factor> -> <unsigned constant> | <variable> | $LPAR <simple expression> $RPAR
25          * <unsigned constant>-> <unsigned number>
26          * <unsigned number>-> $FLOAT | $INTTYPE
27          * <identifier> -> $IDENTIFIER
28          */
29         static void Main(string[] args)
30         {
31             // Provided GOOD test file
32             string inputFilePath = @"C:\projects\CS4100_Compiler_Design\TestInput\GoodtreeA.txt";
33
34             // Provided BAD test file with syntax error
35             // string inputFilePath = @"C:\projects\CS4100_Compiler_Design\TestInput\BadProg1.txt";
36
37             // Provided BAD test file with lexical and syntax error
38             // string inputFilePath = @"C:\projects\CS4100_Compiler_Design\TestInput\BadProg2.txt";
39         }
40     }
41 }
```

```
40      // Initialize structures
41      ReserveTable reserveWords = InitializeReserveWordTable();
42      ReserveTable tokenCodes = InitializeTokenCodeTable();
43      SymbolTable symbolTable = new SymbolTable();
44
45      try
46      {
47          // Initialize input file
48          string[] fileText = InitializeInputFile(inputFilePath);
49
50          // Initialize the Lexical Analyzer (Scanner)
51          LexicalAnalyzer scanner = new LexicalAnalyzer();
52
53          scanner.Initialize(fileText, symbolTable, reserveWords);
54          bool echoOn = true;
55
56          SyntaxAnalyzer parser = new SyntaxAnalyzer(scanner, tokenCodes, echoOn);
57
58          scanner.GetNextToken(echoOn);
59          parser.TraceOn = true;
60          int val = parser.Program();
61
62          symbolTable.PrintSymbolTable();
63      }
64      catch (Exception e)
65      {
66          Console.WriteLine(e.Message);
67      }
68  }
69
70  /// <summary>
71  /// Initializes the reserve table containing the token codes and mnemonics
72  /// </summary>
73  /// <returns>Reserve table containing the token codes and mnemonics</returns>
74  static ReserveTable InitializeTokenCodeTable()
75  {
76      ReserveTable tokenCodes = new ReserveTable();
77
78      // Reserve Words
```

```
79         tokenCodes.Add("GOTO", 0);
80         tokenCodes.Add("_INT", 1);
81         tokenCodes.Add("__TO", 2);
82         tokenCodes.Add("__DO", 3);
83         tokenCodes.Add("__IF", 4);
84         tokenCodes.Add("THEN", 5);
85         tokenCodes.Add("ELSE", 6);
86         tokenCodes.Add("_FOR", 7);
87         tokenCodes.Add("__OF", 8);
88         tokenCodes.Add("WTLN", 9);
89         tokenCodes.Add("RDLN", 10);
90         tokenCodes.Add("_BEG", 11);
91         tokenCodes.Add("_END", 12);
92         tokenCodes.Add("_VAR", 13);
93         tokenCodes.Add("WHIL", 14);
94         tokenCodes.Add("UNIT", 15);
95         tokenCodes.Add("LABL", 16);
96         tokenCodes.Add("REPT", 17);
97         tokenCodes.Add("UNTL", 18);
98         tokenCodes.Add("PROC", 19);
99         tokenCodes.Add("DOWN", 20);
100        tokenCodes.Add("FUNC", 21);
101        tokenCodes.Add("RTRN", 22);
102        tokenCodes.Add("REAL", 23);
103        tokenCodes.Add("_STR", 24);
104        tokenCodes.Add("ARRY", 25);
105
106        // Other Tokens
107        tokenCodes.Add("_DIV", 30);
108        tokenCodes.Add("_MUL", 31);
109        tokenCodes.Add("_ADD", 32);
110        tokenCodes.Add("_SUB", 33);
111        tokenCodes.Add("LPAR", 34);
112        tokenCodes.Add("RPAR", 35);
113        tokenCodes.Add("SEMI", 36);
114        tokenCodes.Add("ASGN", 37);
115        tokenCodes.Add("__GT", 38);
```

```
116         tokenCodes.Add("__LT", 39);
117         tokenCodes.Add("GTEQ", 40);
118         tokenCodes.Add("LTEQ", 41);
119         tokenCodes.Add("__EQ", 42);
120         tokenCodes.Add("NTEQ", 43);
121         tokenCodes.Add("COMM", 44);
122         tokenCodes.Add("LBRC", 45);
123         tokenCodes.Add("RBRC", 46);
124         tokenCodes.Add("COLN", 47);
125         tokenCodes.Add("_DOT", 48);
126
127         // Identifiers
128         tokenCodes.Add("IDNT", 50);
129
130         // Numeric Constants
131         tokenCodes.Add("INTC", 51);
132         tokenCodes.Add("FLTC", 52);
133
134         // String
135         tokenCodes.Add("STRC", 53);
136
137         // Used for any other input characters which are not defined.
138         tokenCodes.Add("UNDF", 99);
139
140         return tokenCodes;
141     }
142
143     /// <summary>
144     /// Initializes reserve table with reserve words and token codes
145     /// </summary>
146     /// <returns>Reserve table with reserve words and token codes</returns>
147     static ReserveTable InitializeReserveWordTable()
148     {
149         ReserveTable reserveWords = new ReserveTable();
150
151         // Token Codes
152         reserveWords.Add("GOTO", 0);
153         reserveWords.Add("INTEGER", 1);
154         reserveWords.Add("TO", 2);
```

```
155     reserveWords.Add("DO", 3);
156     reserveWords.Add("IF", 4);
157     reserveWords.Add("THEN", 5);
158     reserveWords.Add("ELSE", 6);
159     reserveWords.Add("FOR", 7);
160     reserveWords.Add("OF", 8);
161     reserveWords.Add("WRITELN", 9);
162     reserveWords.Add("READLN", 10);
163     reserveWords.Add("BEGIN", 11);
164     reserveWords.Add("END", 12);
165     reserveWords.Add("VAR", 13);
166     reserveWords.Add("WHILE", 14);
167     reserveWords.Add("UNIT", 15);
168     reserveWords.Add("LABEL", 16);
169     reserveWords.Add("REPEAT", 17);
170     reserveWords.Add("UNTIL", 18);
171     reserveWords.Add("PROCEDURE", 19);
172     reserveWords.Add("DOWNT0", 20);
173     reserveWords.Add("FUNCTION", 21);
174     reserveWords.Add("RETURN", 22);
175     reserveWords.Add("REAL", 23);
176     reserveWords.Add("STRING", 24);
177     reserveWords.Add("ARRAY", 25);
178
179     // Other Tokens
180     reserveWords.Add("/", 30);
181     reserveWords.Add("*", 31);
182     reserveWords.Add("+", 32);
183     reserveWords.Add("-", 33);
184     reserveWords.Add("(", 34);
185     reserveWords.Add(")", 35);
186     reserveWords.Add(";", 36);
187     reserveWords.Add(":=", 37);
188     reserveWords.Add(">", 38);
189     reserveWords.Add("<", 39);
190     reserveWords.Add(">=", 40);
191     reserveWords.Add("<=", 41);
```

```
192         reserveWords.Add("=", 42);
193         reserveWords.Add("<>", 43);
194         reserveWords.Add(",", 44);
195         reserveWords.Add("[", 45);
196         reserveWords.Add("]", 46);
197         reserveWords.Add(":", 47);
198         reserveWords.Add(".", 48);
199
200         return reserveWords;
201     }
202
203     /// <summary>
204     /// Reads all the text from the source file and stores each line as a seperate element in a string array.
205     /// </summary>
206     /// <param name="filePath">Path to the file to be read into memory</param>
207     /// <returns>The source text as a string array</returns>
208     static string[] InitializeInputFile(string filePath)
209     {
210         return File.ReadAllLines(filePath);
211     }
212 }
213 }
```

```
1 using System;
2 using System.Collections.Generic;
3 using System.ComponentModel.Design;
4 using System.IO;
5 using System.Linq.Expressions;
6 using System.Security.Cryptography.X509Certificates;
7 using System.Text;
8
9 namespace KyleBushCompiler
10 {
11     class SyntaxAnalyzer
12     {
13         #region Token Constants
14         private const int GOTO = 0;
15         private const int INTEGER = 1;
16         private const int TO = 2;
17         private const int DO = 3;
18         private const int IF = 4;
19         private const int THEN = 5;
20         private const int ELSE = 6;
21         private const int FOR = 7;
22         private const int OF = 8;
23         private const int WRITELN = 9;
24         private const int READLN = 10;
25         private const int BEGIN = 11;
26         private const int END = 12;
27         private const int VAR = 13;
28         private const int WHILE = 14;
29         private const int UNIT = 15;
30         private const int LABEL = 16;
31         private const int REPEAT = 17;
32         private const int UNTIL = 18;
33         private const int PROCEDURE = 19;
34         private const int DOWNT0 = 20;
35         private const int FUNCTION = 21;
36         private const int RETURN = 22;
37         private const int REAL = 23;
```

```
38     private const int STRING = 24;
39     private const int ARRAY = 25;
40     private const int DIVIDE = 30;
41     private const int MULTIPLY = 31;
42     private const int PLUS = 32;
43     private const int MINUS = 33;
44     private const int LPAR = 34;
45     private const int RPAR = 35;
46     private const int SEMICOLON = 36;
47     private const int COLON_EQUALS = 37;
48     private const int GREATER_THAN = 38;
49     private const int LESS_THAN = 39;
50     private const int GREATER_THAN_OR_EQUAL = 40;
51     private const int LESS_THAN_OR_EQUAL = 41;
52     private const int EQUAL = 42;
53     private const int NOT = 43;
54     private const int COMMA = 44;
55     private const int LEFT_BRACKET = 45;
56     private const int RIGHT_BRACKET = 46;
57     private const int COLON = 47;
58     private const int PERIOD = 48;
59     private const int IDENTIFIER = 50;
60     private const int INTTYPE = 51;
61     private const int FLOAT = 52;
62     private const int STRINGTYPE = 53;
63     private const int UNDEFINED = 99;
64     #endregion
65
66     #region Properties
67     public bool TraceOn { get; set; }
68     public bool IsError { get; set; }
69     private LexicalAnalyzer Scanner { get; set; }
70     private ReserveTable TokenCodes { get; set; }
71     private bool ScannerEchoOn { get; set; }
72     private bool Verbose { get; set; }
73
74     #endregion
```



```
75
76     public SyntaxAnalyzer(LexicalAnalyzer scanner, ReserveTable tokenCodes, bool scannerEchoOn)
77     {
78         Scanner = scanner;
79         ScannerEchoOn = scannerEchoOn;
80         TokenCodes = tokenCodes;
81     }
82
83     #region CFG Methods
84
85     /// <summary>
86     /// Implements CFG Rule: <program> -> $UNIT <prog-identifier> $SEMICOLON <block> $PERIOD
87     /// </summary>
88     /// <returns></returns>
89     public int Program()
90     {
91         if (IsError)
92             return -1;
93
94         Debug(true, "Program()");
95
96         if (Scanner.TokenCode == UNIT)
97         {
98             GetNextToken();
99             int x = ProgIdentifier();
100             if (Scanner.TokenCode == SEMICOLON)
101             {
102                 GetNextToken();
103                 x = Block();
104                 if (Scanner.TokenCode == PERIOD)
105                 {
106                     GetNextToken();
107                 }
108                 else
109                 {
110                     Error("PERIOD");
111                 }
112             }
113             else
114             {
```

```
115         Error("SEMICOLON");
116     }
117 }
118 else
119 {
120     Error("UNIT");
121 }
122
123 Debug(false, "Program()");
124 return -1;
125 }
126
127 /// <summary>
128 /// Implements CFG Rule: <block> -> $BEGIN <statement> {$SEMICOLON <statement>}* $END
129 /// </summary>
130 /// <returns></returns>
131 private int Block()
132 {
133     if (IsError)
134         return -1;
135
136     Debug(true, "Block()");
137     if (Scanner.TokenCode == BEGIN)
138     {
139         GetNextToken();
140         int x = Statement();
141         while (Scanner.TokenCode == SEMICOLON && !IsError)
142         {
143             GetNextToken();
144             x = Statement();
145         }
146
147         if (Scanner.TokenCode == END)
148             GetNextToken();
149         else
150             Error("END");
151     }
152     else
153         Error("BEGIN");
154 }
```

```
155         Debug(false, "Block()");
156         return -1;
157     }
158
159     /// <summary>
160     /// Implements CFG Rule: <prog-identifier> -> <identifier>
161     /// </summary>
162     /// <returns></returns>
163     private int ProgIdentifier()
164     {
165         if (IsError)
166             return -1;
167
168         Debug(true, "ProgIdentifier()");
169         Identifier();
170         Debug(false, "ProgIdentifier()");
171         return -1;
172     }
173
174     /// <summary>
175     /// Implements CFG Rule: <statement> -> <variable> $COLON_EQUALS <simple expression>
176     /// </summary>
177     /// <returns></returns>
178     private int Statement()
179     {
180         if (IsError)
181             return -1;
182
183         Debug(true, "Statement()");
184         int x = Variable();
185         if (Scanner.TokenCode == COLON_EQUALS)
186         {
187             GetNextToken();
188             x = SimpleExpression();
189         }
190         else
191             Error("COLON-EQUALS");
192
193         Debug(false, "Statement()");
```

```
194         return -1;
195     }
196
197     /// <summary>
198     /// Implements CFG Rule: <variable> -> <identifier>
199     /// </summary>
200     /// <returns></returns>
201     private int Variable()
202     {
203         if (IsError)
204             return -1;
205
206         Debug(true, "Variable()");
207         Identifier();
208         Debug(false, "Variable()");
209         return -1;
210     }
211
212     /// <summary>
213     /// Implements CFG Rule: <simple expression> -> [<sign>] <term> {<addop> <term>}*
214     /// </summary>
215     /// <returns></returns>
216     private int SimpleExpression()
217     {
218         if (IsError)
219             return -1;
220
221         Debug(true, "SimpleExpression()");
222
223         int x;
224
225         if (isSign())
226         {
227             x = Sign();
228         }
229
230         x = Term();
231
232         while (isAddOp() && !IsError)
233
```

```
234         x = AddOp();
235         x = Term();
236     }
237
238     Debug(false, "SimpleExpression()");
239     return -1;
240 }
241
242 /// <summary>
243 /// Implements CFG Rule: <addop> -> $PLUS | $MINUS
244 /// </summary>
245 /// <returns></returns>
246 private int AddOp()
247 {
248     if (IsError)
249         return -1;
250
251     Debug(true, "AddOp()");
252     if (Scanner.TokenCode == PLUS || Scanner.TokenCode == MINUS)
253         GetNextToken();
254     else
255         Error("PLUS or MINUS");
256     Debug(false, "AddOp()");
257     return -1;
258 }
259
260 /// <summary>
261 /// Checks if the next token is an AddOp token.
262 /// </summary>
263 /// <returns></returns>
264 private bool isAddOp()
265 {
266     if (Scanner.TokenCode == PLUS || Scanner.TokenCode == MINUS)
267         return true;
268     else
269         return false;
270 }
271
272 /// <summary>
```

```
273     /// Implements CFG Rule: <sign> -> $PLUS | $MINUS
274     /// </summary>
275     /// <returns></returns>
276     private int Sign()
277     {
278         if (IsError)
279             return -1;
280
281         Debug(true, "Sign()");
282         if (Scanner.TokenCode == PLUS)
283             GetNextToken();
284         else if (Scanner.TokenCode == MINUS)
285             GetNextToken();
286         else
287             Error("PLUS or MINUS");
288         Debug(false, "Sign()");
289         return -1;
290     }
291
292     /// <summary>
293     /// Checks if the next token is a Sign token.
294     /// </summary>
295     /// <returns></returns>
296     private bool isSign()
297     {
298         if (Scanner.TokenCode == PLUS || Scanner.TokenCode == MINUS)
299             return true;
300         else
301             return false;
302     }
303
304     /// <summary>
305     /// Implements CFG Rule: <term> -> <factor> {<mulop> <factor> }*
306     /// </summary>
307     /// <returns></returns>
308     private int Term()
309     {
310         if (IsError)
```

```
311         return -1;
312
313         Debug(true, "Term()");
314         int x = Factor();
315
316         while (isMulOp() && !IsError)
317         {
318             x = MulOp();
319             x = Factor();
320         }
321
322         Debug(false, "Term()");
323         return -1;
324     }
325
326     /// <summary>
327     /// Implements CFG Rule: <mulop> -> $MULTIPLY | $DIVIDE
328     /// </summary>
329     /// <returns></returns>
330     private int MulOp()
331     {
332         if (IsError)
333             return -1;
334
335         Debug(true, "MulOp()");
336
337         if (Scanner.TokenCode == MULTIPLY || Scanner.TokenCode == DIVIDE)
338             GetNextToken();
339         else
340             Error("MULTIPLY or DIVIDE");
341
342         Debug(false, "MulOp()");
343         return -1;
344     }
345
346     /// <summary>
347     /// Checks if the next token is a MulOp token.
348     /// </summary>
349     /// <returns></returns>
```

```
350     private bool isMulOp()
351     {
352         if (Scanner.TokenCode == MULTIPLY || Scanner.TokenCode == DIVIDE)
353             return true;
354         else
355             return false;
356     }
357
358
359     /// <summary>
360     /// Implements CFG Rule: <factor> -> <unsigned constant> | <variable> | $LPAR <simple expression> $RPAR
361     /// </summary>
362     /// <returns></returns>
363     private int Factor()
364     {
365         if (IsError)
366             return -1;
367
368         Debug(true, "Factor()");
369
370         int x;
371
372         if (isUnsignedConstant())
373         {
374             x = UnsignedConstant();
375         }
376         else if (isVariable())
377         {
378             Variable();
379         }
380         else if (Scanner.TokenCode == LPAR)
381         {
382             GetNextToken();
383             SimpleExpression();
384             if (Scanner.TokenCode == RPAR)
385                 GetNextToken();
386             else
387                 Error("RPAR");
388         }
```



```
389         else
390             Error("UNSIGNED CONSTANT or VARIABLE or LPAR");
391
392         Debug(false, "Factor()");
393         return -1;
394     }
395
396
397
398     /// <summary>
399     /// Checks if the next token is an Unsigned Constant
400     /// </summary>
401     /// <returns></returns>
402     private bool isUnsignedConstant()
403     {
404         if (Scanner.TokenCode == FLOAT || Scanner.TokenCode == INTTYPE)
405             return true;
406         else
407             return false;
408     }
409
410     /// <summary>
411     /// Implements CFG Rule: <unsigned constant>-> <unsigned number>
412     /// </summary>
413     /// <returns></returns>
414     private int UnsignedConstant()
415     {
416         if (IsError)
417             return -1;
418
419         Debug(true, "UnsignedConstant()");
420         UnsignedNumber();
421         Debug(false, "UnsignedConstant()");
422         return -1;
423     }
424
425     /// <summary>
426     /// Implements CFG Rule: <unsigned number>-> $FLOAT | $INTTYPE
427     /// </summary>
```

```
428     /// <returns></returns>
429     private int UnsignedNumber()
430     {
431         if (IsError)
432             return -1;
433
434         Debug(true, "UnsignedNumber()");
435
436         if (Scanner.TokenCode == FLOAT || Scanner.TokenCode == INTTYPE)
437             GetNextToken();
438         else
439             Error("FLOAT or INTTYPE");
440
441         Debug(false, "UnsignedNumber()");
442         return -1;
443     }
444
445     /// <summary>
446     /// Checks if the next token is a Variable
447     /// </summary>
448     /// <returns></returns>
449     private bool isVariable()
450     {
451         if (Scanner.TokenCode == IDENTIFIER)
452             return true;
453         else
454             return false;
455     }
456
457     /// <summary>
458     /// Implements CFG Rule: <identifier> -> $IDENTIFIER
459     /// </summary>
460     /// <returns></returns>
461     private int Identifier()
462     {
463         if (IsError)
464             return -1;
465
466         Debug(true, "Identifier()");
```

```
467
468         if (Scanner.TokenCode == IDENTIFIER)
469             GetNextToken();
470         else
471             Error("IDENTIFIER");
472
473         Debug(false, "Identifier()");
474         return -1;
475     }
476
477     #endregion
478
479     #region Utility Methods
480
481     /// <summary>
482     /// Prints an error with the expected token type and the actual token found.
483     /// </summary>
484     /// <param name="expectedToken">The expected token type.</param>
485     private void Error(string expectedToken)
486     {
487         IsError = true;
488         Console.WriteLine("Line #{0}: {1}", Scanner.CurrentLineIndex + 1, Scanner.CurrentLine);
489         Console.WriteLine("ERROR: {0} expected, but {1} found.", expectedToken, Scanner.NextToken);
490     }
491
492     /// <summary>
493     /// Prints the method that is being entered or exited if TraceOn is set to true
494     /// </summary>
495     /// <param name="entering"></param>
496     /// <param name="name"></param>
497     private void Debug(bool entering, string name)
498     {
499         if (TraceOn)
500         {
501             if (entering)
502                 Console.WriteLine("ENTERING " + name);
503             else
504                 Console.WriteLine("EXITING " + name);
505         }
506     }
507 }
```

```
506     }
507 }
508
509 /// <summary>
510 /// Gets the next token and prints the token lexeme and mnemonic if Trace is on.
511 /// </summary>
512 private void GetNextToken()
513 {
514     Scanner.GetNextToken(ScannerEchoOn);
515     if (TraceOn)
516         Console.WriteLine("Lexeme: {0} Mnemonic: {1}", Scanner.NextToken, TokenCodes.LookupCode
                             (Scanner.TokenCode));
517 }
518
519 #endregion
520 }
521 }
522
```