```
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 DOWNTO = 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 ASSIGN = 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_EQUAL = 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          public bool ErrorOcurred { get; set; }
70          private bool PrintError { get; set; }
71          private LexicalAnalyzer Scanner { get; set; }
72          private ReserveTable TokenCodes { get; set; }
73          private bool ScannerEchoOn { get; set; }
74          private bool Verbose { get; set; }
```

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

```
114                         GetNextToken();
115                     }
116                     else
117                     {
118                         UnexpectedTokenError("PERIOD");
119                     }
120                 }
121                 else
122                 {
123                     UnexpectedTokenError("SEMICOLON");
124                 }
125             }
126             else
127             {
128                 UnexpectedTokenError("UNIT");
129             }
130 
131             Debug(false, "Program()");
132             return -1;
133         }
134 
135 
136         /// <summary>
137         /// Implements CFG Rule: <block> -> [<label-declaration>] {<variable-dec-sec>}* <block-body>
138         /// Also contains main error handling logic.
139         /// </summary>
140         /// <returns></returns>
141         private int Block()
142         {
143             if (IsError)
144                 return -1;
145 
146             Debug(true, "Block()");
147             if (Scanner.TokenCode == LABEL)
148             {
149                 LabelDeclaration();
150             }
151 
152             while (Scanner.TokenCode == VAR && !IsError)
153             {
```

```
154                 VariableDecSec();
155             }
156
157             BlockBody();
158
159             // Error handling and resyncing
160             while (IsError == true && !Scanner.EndOfFile)
161             {
162                 Resync();
163                 IsError = false;
164                 PrintError = true;
165                 while (IsError == false && !Scanner.EndOfFile)
166                 {
167                     Statement();
168
169                     if (Scanner.TokenCode == END)
170                     {
171                         GetNextToken();
172                         if (Scanner.TokenCode == PERIOD)
173                             GetNextToken();
174                         else
175                             UnexpectedTokenError("PERIOD");
176                     }
177                     else if (Scanner.TokenCode == SEMICOLON)
178                         GetNextToken();
179                     else
180                         UnexpectedTokenError("END or SEMICOLON");
181                 }
182             }
183
184             Debug(false, "Block()");
185             return -1;
186         }
187
188         /// <summary>
189         /// Implements CFG Rule: <block-body> -> $BEGIN <statement> {$SEMICOLON <statement>}* $END
190         /// </summary>
191         /// <returns></returns>
192         private int BlockBody()
```

```
193         {
194             if (IsError)
195                 return -1;
196 
197             Debug(true, "BlockBody()");
198             if (Scanner.TokenCode == BEGIN)
199             {
200                 GetNextToken();
201                 int x = Statement();
202                 while (Scanner.TokenCode == SEMICOLON && !IsError)
203                 {
204                     GetNextToken();
205                     x = Statement();
206                 }
207 
208                 if (Scanner.TokenCode == END)
209                     GetNextToken();
210                 else
211                     UnexpectedTokenError("END or SEMICOLON");
212             }
213             else
214                 UnexpectedTokenError("BEGIN");
215 
216             Debug(false, "BlockBody()");
217             return -1;
218         }
219 
220         /// <summary>
221         /// Implements CFG Rule: <label-declaration> -> $LABEL <identifier> {$COMMA <identifier>}* $SEMICOLON
222         /// </summary>
223         /// <returns></returns>
224         private int LabelDeclaration()
225         {
226             if (IsError)
227                 return -1;
228 
229             Debug(true, "LabelDeclaration()");
230             if (Scanner.TokenCode == LABEL)
231             {
```

```
232                 GetNextToken();
233                 if (Scanner.TokenCode == IDENTIFIER)
234                 {
235                     if (isNotPreviouslyDeclaredIdentifier(SymbolKind.Label))
236                     {
237                         int x = Identifier();
238 
239 
240                         while (Scanner.TokenCode == COMMA && !IsError)
241                         {
242                             GetNextToken();
243                             if (Scanner.TokenCode == IDENTIFIER)
244                             {
245                                 if (isNotPreviouslyDeclaredIdentifier(SymbolKind.Label))
246                                 {
247                                     x = Identifier();
248                                 }
249                             }
250                         }
251 
252                         if (Scanner.TokenCode == SEMICOLON)
253                             GetNextToken();
254                         else
255                             UnexpectedTokenError("SEMICOLON");
256                     }
257                 }
258             }
259             else
260                 UnexpectedTokenError("LABEL");
261 
262             Debug(false, "LabelDeclaration()");
263             return -1;
264         }
265 
266 
267 
268         /// <summary>
269         /// Implements CFG Rule: <variable-dec-sec> -> $VAR <variable-declaration>
270         /// </summary>
271         /// <returns></returns>
```

```
272         private int VariableDecSec()
273         {
274             if (IsError)
275                 return -1;
276 
277             Debug(true, "VariableDecSec()");
278             if (Scanner.TokenCode == VAR)
279             {
280                 GetNextToken();
281                 int x = VariableDeclaration();
282             }
283             else
284                 UnexpectedTokenError("VAR");
285 
286             Debug(false, "VariableDecSec()");
287             return -1;
288         }
289 
290         /// <summary>
291         /// Implements CFG Rule: <variable-declaration> -> {<identifier> {$COMMA <identifier>}* $COLON <type>
              $SEMICOLON}+
292         /// </summary>
293         /// <returns></returns>
294         private int VariableDeclaration()
295         {
296             List<string> variables = new List<string>();
297             string type = "";
298 
299             if (IsError)
300                 return -1;
301 
302             Debug(true, "VariableDeclaration()");
303 
304             if (DeclaredLabels.Contains(Scanner.NextToken))
305             {
306                 RedeclaredIdentifierError("LABEL", "VARIABLE");
307             }
308             else if (ProgramName == Scanner.NextToken)
309             {
```

```
310                 RedeclaredIdentifierError("ProgramName", "VARIABLE");
311             }
312             else
313             {
314                 do
315                 {
316                     variables.Add(Scanner.NextToken);
317                     DeclaredVariables.Add(Scanner.NextToken);
318                     int x = Variable();
319                     while (Scanner.TokenCode == COMMA && !IsError)
320                     {
321                         GetNextToken();
322                         variables.Add(Scanner.NextToken);
323                         DeclaredVariables.Add(Scanner.NextToken);
324                         x = Variable();
325                     }
326                     if (Scanner.TokenCode == COLON)
327                     {
328                         GetNextToken();
329                         type = Scanner.NextToken;
330                         x = Type();
331                         if (Scanner.TokenCode == SEMICOLON)
332                         {
333                             GetNextToken();
334                         }
335                         else
336                         {
337                             UnexpectedTokenError("SEMICOLON");
338                         }
339
340                         SetVariableType(variables, type);
341                     }
342                     else
343                     {
344                         UnexpectedTokenError("COMMA");
345                     }
346                 } while (Scanner.TokenCode == IDENTIFIER && !IsError);
347             }
348
349
```

```
350             Debug(false, "VariableDeclaration()");
351             return -1;
352         }
353
354
355
356
357
358
359         /// <summary>
360         /// Implements CFG Rule: <statement>-> {<label> $COLON]}*
361         ///                                     [
362         ///                                         <variable> $ASSIGN (<simple expression> | <string literal>) |
363         ///                                         <block-body> |
364         ///                                         $IF <relexpression> $THEN <statement> [$ELSE <statement>] |
365         ///                                         $WHILE <relexpression> $DO <statement> |
366         ///                                         $REPEAT <statement> $UNTIL <relexpression> |
367         ///                                         $FOR <variable> $ASSIGN <simple expression> $TO <simple
              expression> $DO <statement> |
368         ///                                         $GOTO <label> |
369         ///                                         $WRITELN $LPAR (<simple expression> | <identifier> |
              <stringconst>) $RPAR
370         ///                                     ]+
371         /// </summary>
372         /// <returns></returns>
373         private int Statement()
374         {
375             if (IsError)
376                 return -1;
377
378             Debug(true, "Statement()");
379             while (IsLabel() && !IsError)
380             {
381                 int x = Label();
382                 if (Scanner.TokenCode == COLON)
383                     GetNextToken();
384             }
385             if (isVariable())
386             {
```

```
387                 Variable();
388                 if (Scanner.TokenCode == ASSIGN)
389                 {
390                     GetNextToken();
391                     if (IsSimpleExpression())
392                         SimpleExpression();
393                     else if (Scanner.TokenCode == STRINGTYPE)
394                         StringConst();
395                     else
396                         UnexpectedTokenError("SIMPLE EXPRESSION or STRING");
397                 }
398                 else
399                     UnexpectedTokenError("IDENTIFIER");
400             }
401             else if (Scanner.TokenCode == BEGIN)
402             {
403                 BlockBody();
404             }
405             else if (Scanner.TokenCode == IF)
406             {
407                 GetNextToken();
408                 RelExpression();
409                 if (Scanner.TokenCode == THEN)
410                 {
411                     GetNextToken();
412                     Statement();
413                     if (Scanner.TokenCode == ELSE)
414                     {
415                         GetNextToken();
416                         Statement();
417                     }
418                 }
419                 else
420                     UnexpectedTokenError("THEN");
421             }
422             else if (Scanner.TokenCode == WHILE)
423             {
424                 GetNextToken();
425                 RelExpression();
```

```
426                 if (Scanner.TokenCode == DO)
427                 {
428                     GetNextToken();
429                     Statement();
430                 }
431                 else
432                     UnexpectedTokenError("DO");
433             }
434             else if (Scanner.TokenCode == REPEAT)
435             {
436                 GetNextToken();
437                 Statement();
438                 if (Scanner.TokenCode == UNTIL)
439                 {
440                     GetNextToken();
441                     RelExpression();
442                 }
443                 else
444                     UnexpectedTokenError("UNTIL");
445             }
446             else if (Scanner.TokenCode == FOR)
447             {
448                 GetNextToken();
449                 Variable();
450                 if (Scanner.TokenCode == ASSIGN)
451                 {
452                     GetNextToken();
453                     SimpleExpression();
454                     if (Scanner.TokenCode == TO)
455                     {
456                         GetNextToken();
457                         SimpleExpression();
458                         if (Scanner.TokenCode == DO)
459                         {
460                             GetNextToken();
461                             Statement();
462                         }
463                         else
464                             UnexpectedTokenError("DO");
```

```
465                     }
466                     else
467                         UnexpectedTokenError("TO");
468                 }
469                 else
470                     UnexpectedTokenError("ASSIGN");
471             }
472             else if (Scanner.TokenCode == GOTO)
473             {
474                 GetNextToken();
475                 Label();
476             }
477             else if (Scanner.TokenCode == WRITELN)
478             {
479                 GetNextToken();
480                 if (Scanner.TokenCode == LPAR)
481                 {
482                     GetNextToken();
483                     if (IsSimpleExpression())
484                     {
485                         SimpleExpression();
486                         if (Scanner.TokenCode == RPAR)
487                             GetNextToken();
488                         else
489                             UnexpectedTokenError("RPAR");
490                     }
491                     else if (Scanner.TokenCode == IDENTIFIER)
492                     {
493                         Identifier();
494                         if (Scanner.TokenCode == RPAR)
495                             GetNextToken();
496                         else
497                             UnexpectedTokenError("RPAR");
498                     }
499                     else if (Scanner.TokenCode == STRINGTYPE)
500                     {
501                         StringConst();
502                         if (Scanner.TokenCode == RPAR)
503                             GetNextToken();
```

```
504                         else
505                             UnexpectedTokenError("RPAR");
506                     }
507                     else
508                         UnexpectedTokenError("SimpleExpression or IDENTIFIER or STRINGTYPE");
509                 }
510                 else
511                     UnexpectedTokenError("LPAR");
512             }
513             else
514                 UnexpectedTokenError("Statement Token");
515 
516             Debug(false, "Statement()");
517             return -1;
518         }
519 
520         /// <summary>
521         /// Implements CFG Rule: <prog-identifier> -> <identifier>
522         /// </summary>
523         /// <returns></returns>
524         private int ProgIdentifier()
525         {
526             if (IsError)
527                 return -1;
528 
529             Debug(true, "ProgIdentifier()");
530             UpdateSymbolKind(SymbolKind.ProgName);
531             ProgramName = Scanner.NextToken;
532             Identifier();
533             Debug(false, "ProgIdentifier()");
534             return -1;
535         }
536 
537         /// <summary>
538         /// Implements CFG Rule: <variable> -> <identifier> [$LEFT_BRACKET <simple expression> $RIGHT_BRACKET]
539         /// </summary>
540         /// <returns></returns>
541         private int Variable()
542         {
```

```
543             if (IsError)
544                 return -1;
545
546             Debug(true, "Variable()");
547
548             int index = Scanner.SymbolTable.LookupSymbol(Scanner.NextToken);
549             if (index != -1)
550             {
551                 Symbol symbol = Scanner.SymbolTable.GetSymbol(index);
552                 if (symbol.Kind == SymbolKind.Variable)
553                 {
554                     if (DeclaredLabels.Contains(Scanner.NextToken))
555                     {
556                         DeclarationWarning(SymbolKind.Variable, SymbolKind.Label);
557                         DeclaredVariables.Add(Scanner.NextToken);
558                     }
559                     else if (!DeclaredVariables.Contains(Scanner.NextToken))
560                     {
561                         UndeclaredWarning();
562                         DeclaredVariables.Add(Scanner.NextToken);
563                     }
564
565                 }
566                 else
567                     DeclarationWarning(SymbolKind.Variable, symbol.Kind);
568
569                 Identifier();
570
571
572                 if (Scanner.TokenCode == LEFT_BRACKET)
573                 {
574                     GetNextToken();
575                     SimpleExpression();
576                     if (Scanner.TokenCode == RIGHT_BRACKET)
577                         GetNextToken();
578                     else
579                         UnexpectedTokenError("RIGHT_BRACKET");
580                 }
581             }
582
```

```
583             Debug(false, "Variable()");
584             return -1;
585         }
586 
587         /// <summary>
588         /// Implements CFG Rule: <label> -> <identifier>
589         /// </summary>
590         /// <returns></returns>
591         private int Label()
592         {
593             if (IsError)
594                 return -1;
595 
596             Debug(true, "Label()");
597             // Checks that the indentifier has been declared as type label
598             if (IsLabel())
599                 Identifier();
600             else
601                 UnexpectedTokenError("LABEL");
602 
603             Debug(false, "Label()");
604             return -1;
605         }
606 
607         /// <summary>
608         /// Implements CFG Rule: <relexpression> -> <simple expression> <relop> <simple expression>
609         /// </summary>
610         /// <returns></returns>
611         private int RelExpression()
612         {
613             if (IsError)
614                 return -1;
615 
616             Debug(true, "Label()");
617             SimpleExpression();
618             RelOp();
619             SimpleExpression();
620             Debug(false, "Label()");
621 
```

```
622         }
623
624         /// <summary>
625         /// Implements CFG Rule: <relop> -> $EQ | $LSS | $GTR | $NEQ | $LEQ | $GEQ
626         /// </summary>
627         /// <returns></returns>
628         private int RelOp()
629         {
630             if (IsError)
631                 return -1;
632
633             Debug(true, "Label()");
634             switch (Scanner.TokenCode)
635             {
636                 case EQUAL:
637                 case LESS_THAN:
638                 case GREATER_THAN:
639                 case LESS_THAN_OR_EQUAL:
640                 case GREATER_THAN_OR_EQUAL:
641                 case NOT_EQUAL:
642                     GetNextToken();
643                     break;
644                 default:
645                     UnexpectedTokenError("Relational Operator");
646                     break;
647             }
648             Debug(false, "Label()");
649             return -1;
650         }
651
652         /// <summary>
653         /// Implements CFG Rule: <simple expression> -> [<sign>] <term> {<addop> <term>}*
654         /// </summary>
655         /// <returns></returns>
656         private int SimpleExpression()
657         {
658             if (IsError)
659                 return -1;
660
```

```
661             Debug(true, "SimpleExpression()");
662 
663             int x;
664 
665             if (isSign())
666             {
667                 x = Sign();
668             }
669 
670             x = Term();
671 
672             while (isAddOp() && !IsError)
673             {
674                 x = AddOp();
675                 x = Term();
676             }
677 
678             Debug(false, "SimpleExpression()");
679             return -1;
680         }
681 
682         /// <summary>
683         /// Implements CFG Rule: <addop> -> $PLUS | $MINUS
684         /// </summary>
685         /// <returns></returns>
686         private int AddOp()
687         {
688             if (IsError)
689                 return -1;
690 
691             Debug(true, "AddOp()");
692             if (Scanner.TokenCode == PLUS || Scanner.TokenCode == MINUS)
693                 GetNextToken();
694             else
695                 UnexpectedTokenError("PLUS or MINUS");
696             Debug(false, "AddOp()");
697             return -1;
698         }
699 
700 
```

```
701
702         /// <summary>
703         /// Implements CFG Rule: <sign> -> $PLUS | $MINUS
704         /// </summary>
705         /// <returns></returns>
706         private int Sign()
707         {
708             if (IsError)
709                 return -1;
710
711             Debug(true, "Sign()");
712             if (Scanner.TokenCode == PLUS)
713                 GetNextToken();
714             else if (Scanner.TokenCode == MINUS)
715                 GetNextToken();
716             else
717                 UnexpectedTokenError("PLUS or MINUS");
718             Debug(false, "Sign()");
719             return -1;
720         }
721
722
723         /// <summary>
724         /// Implements CFG Rule: <term> -> <factor> {<mulop> <factor> }*
725         /// </summary>
726         /// <returns></returns>
727         private int Term()
728         {
729             if (IsError)
730                 return -1;
731
732             Debug(true, "Term()");
733             int x = Factor();
734
735             while (isMulOp() && !IsError)
736             {
737                 x = MulOp();
738                 x = Factor();
739             }
```

```
740
741             Debug(false, "Term()");
742             return -1;
743         }
744
745         /// <summary>
746         /// Implements CFG Rule: <mulop> -> $MULTIPLY | $DIVIDE
747         /// </summary>
748         /// <returns></returns>
749         private int MulOp()
750         {
751             if (IsError)
752                 return -1;
753
754             Debug(true, "MulOp()");
755
756             if (Scanner.TokenCode == MULTIPLY || Scanner.TokenCode == DIVIDE)
757                 GetNextToken();
758             else
759                 UnexpectedTokenError("MULTIPLY or DIVIDE");
760
761             Debug(false, "MulOp()");
762             return -1;
763         }
764
765         /// <summary>
766         /// Implements CFG Rule: <factor> -> <unsigned constant> | <variable> | $LPAR <simple expression> $RPAR
767         /// </summary>
768         /// <returns></returns>
769         private int Factor()
770         {
771             if (IsError)
772                 return -1;
773
774             Debug(true, "Factor()");
775
776             int x;
777
778             if (isUnsignedConstant())
```

```
779             {
780                 x = UnsignedConstant();
781             }
782             else if (isVariable())
783             {
784                 Variable();
785             }
786             else if (Scanner.TokenCode == LPAR)
787             {
788                 GetNextToken();
789                 SimpleExpression();
790                 if (Scanner.TokenCode == RPAR)
791                     GetNextToken();
792                 else
793                     UnexpectedTokenError("RPAR");
794             }
795             else
796                 UnexpectedTokenError("UNSIGNED CONSTANT or VARIABLE or LPAR");
797 
798             Debug(false, "Factor()");
799             return -1;
800         }
801 
802         /// <summary>
803         /// Implements CFG Rule: <type> -> <simple type> | $ARRAY $LBRACK $INTTYPE $RBRACK $OF $INTEGER
804         /// </summary>
805         /// <returns></returns>
806         private int Type()
807         {
808             if (IsError)
809                 return -1;
810 
811             Debug(true, "Type()");
812             if (IsSimpleType())
813                 SimpleType();
814             else if (Scanner.TokenCode == ARRAY)
815             {
816                 GetNextToken();
817                 if (Scanner.TokenCode == LEFT_BRACKET)
```

```
818                 {
819                     GetNextToken();
820                     if (Scanner.TokenCode == INTTYPE)
821                     {
822                         GetNextToken();
823                         if (Scanner.TokenCode == RIGHT_BRACKET)
824                         {
825                             GetNextToken();
826                             if (Scanner.TokenCode == OF)
827                             {
828                                 GetNextToken();
829                                 if (Scanner.TokenCode == INTEGER)
830                                 {
831                                     GetNextToken();
832                                 }
833                                 else
834                                     UnexpectedTokenError("INTEGER");
835                             }
836                             else
837                                 UnexpectedTokenError("OF");
838                         }
839                         else
840                             UnexpectedTokenError("RIGHT_BRACKET");
841                     }
842                     else
843                         UnexpectedTokenError("INTTYPE");
844                 }
845                 else
846                     UnexpectedTokenError("LEFT_BRACKET");
847             }
848             else
849                 UnexpectedTokenError("Simple Type or ARRAY");
850 
851             Debug(false, "Type()");
852             return -1;
853         }
854 
855         /// <summary>
856         /// Implements CFG Rule: <simple type> -> $INTEGER | $FLOAT | $STRING
```

```
857         /// </summary>
858         /// <returns></returns>
859         private int SimpleType()
860         {
861             if (IsError)
862                 return -1;
863 
864             Debug(true, "SimpleType()");
865 
866             if (Scanner.TokenCode == INTEGER || Scanner.TokenCode == REAL || Scanner.TokenCode == STRING)
867                 GetNextToken();
868             else
869                 UnexpectedTokenError("INTEGER or FLOAT or STRING");
870 
871             Debug(false, "SimpleType()");
872             return -1;
873         }
874 
875         /// <summary>
876         /// Implements CFG Rule: <constant> -> [<sign>] <unsigned constant>
877         /// </summary>
878         /// <returns></returns>
879         private int Constant()
880         {
881             if (IsError)
882                 return -1;
883 
884             Debug(true, "Constant()");
885 
886             if (isSign())
887                 Sign();
888             UnsignedConstant();
889 
890             Debug(false, "Constant()");
891             return -1;
892         }
893 
894         /// <summary>
895         /// Implements CFG Rule: <unsigned constant>-> <unsigned number>
```

```
896         /// </summary>
897         /// <returns></returns>
898         private int UnsignedConstant()
899         {
900             if (IsError)
901                 return -1;
902 
903             Debug(true, "UnsignedConstant()");
904             UnsignedNumber();
905             Debug(false, "UnsignedConstant()");
906             return -1;
907         }
908 
909         /// <summary>
910         /// Implements CFG Rule: <unsigned number>-> $FLOAT | $INTTYPE
911         /// </summary>
912         /// <returns></returns>
913         private int UnsignedNumber()
914         {
915             if (IsError)
916                 return -1;
917 
918             Debug(true, "UnsignedNumber()");
919 
920             if (Scanner.TokenCode == FLOAT || Scanner.TokenCode == INTTYPE)
921                 GetNextToken();
922             else
923                 UnexpectedTokenError("FLOAT or INTTYPE");
924 
925             Debug(false, "UnsignedNumber()");
926             return -1;
927         }
928 
929 
930 
931         /// <summary>
932         /// Implements CFG Rule: <identifier> -> $IDENTIFIER
933         /// </summary>
934         /// <returns></returns>
```

```
935         private int Identifier()
936         {
937             if (IsError)
938                 return -1;
939 
940             Debug(true, "Identifier()");
941 
942             if (Scanner.TokenCode == IDENTIFIER)
943                 GetNextToken();
944             else
945                 UnexpectedTokenError("IDENTIFIER");
946 
947             Debug(false, "Identifier()");
948             return -1;
949         }
950 
951         /// <summary>
952         /// Implements CFG Rule: <stringconst> -> $STRINGTYPE
953         /// </summary>
954         /// <returns></returns>
955         private int StringConst()
956         {
957             if (IsError)
958                 return -1;
959 
960             Debug(true, "StringConst()");
961 
962             if (Scanner.TokenCode == STRINGTYPE)
963 
964                 GetNextToken();
965             else
966                 UnexpectedTokenError("STRINGYPE");
967 
968             Debug(false, "StringConst()");
969             return -1;
970         }
971 
972         #endregion
973 
```

```
974         #region Errors and Warnings
975
976         /// <summary>
977         /// Prints an error with the expected token type and the actual token found.
978         /// </summary>
979         /// <param name="expectedToken">The expected token type.</param>
980         private void UnexpectedTokenError(string expectedToken)
981         {
982             IsError = true;
983             ErrorOcurred = true;
984
985             if (PrintError)
986             {
987                 Console.WriteLine("\n********** Error **********");
988                 Console.WriteLine("Line #{0}: {1}", Scanner.CurrentLineIndex, Scanner.CurrentLine);
989                 Console.WriteLine("ERROR: {0} expected, but {1} found.", expectedToken, Scanner.NextToken);
990                 Console.WriteLine("**************************\n");
991             }
992
993             PrintError = false;
994         }
995
996         /// <summary>
997         /// Prints a warning message when an identifier is detected that was undeclared.
998         /// </summary>
999         private void UndeclaredWarning()
1000        {
1001            Console.WriteLine("\n********** Warning **********");
1002            Console.WriteLine("Line #{0}: {1}", Scanner.CurrentLineIndex, Scanner.CurrentLine);
1003            Console.WriteLine("WARNING: {0} undeclared.", Scanner.NextToken);
1004            Console.WriteLine("**************************\n");
1005        }
1006
1007        /// <summary>
1008        /// Prints a warning message when an identifier is used as a different Kind than what it was declared.
1009        /// </summary>
1010        /// <param name="expected"></param>
1011        /// <param name="found"></param>
1012        private void DeclarationWarning(SymbolKind expected, SymbolKind found)
```

```
1013        {
1014            Console.WriteLine("\n********** Warning **********");
1015            Console.WriteLine("Line #{0}: {1}", Scanner.CurrentLineIndex, Scanner.CurrentLine);
1016            Console.WriteLine("WARNING: {0} declared as expected, but used as {1}.", Scanner.NextToken, expected,
                  found);
1017            Console.WriteLine("***************************\n");
1018        }
1019
1020        /// <summary>
1021        /// Displays
1022        /// </summary>
1023        /// <param name="used"></param>
1024        /// <param name="declared"></param>
1025        private void RedeclaredIdentifierError(string used, string declared)
1026        {
1027            IsError = true;
1028            ErrorOcurred = true;
1029
1030            Console.WriteLine("\n********** Error **********");
1031            Console.WriteLine("Line #{0}: {1}", Scanner.CurrentLineIndex, Scanner.CurrentLine);
1032            Console.WriteLine("WARNING: {0} used, but {1} declared.", used, declared);
1033            Console.WriteLine("***************************\n");
1034        }
1035
1036        #endregion
1037
1038        #region Type Testing
1039
1040        /// <summary>
1041        /// Checks if the next token is a Sign token.
1042        /// </summary>
1043        /// <returns></returns>
1044        private bool isSign()
1045        {
1046            if (Scanner.TokenCode == PLUS || Scanner.TokenCode == MINUS)
1047                return true;
1048            else
1049                return false;
1050        }
```

```
1051
1052            /// <summary>
1053            /// Checks if the next token is an AddOp token.
1054            /// </summary>
1055            /// <returns></returns>
1056            private bool isAddOp()
1057            {
1058                if (Scanner.TokenCode == PLUS || Scanner.TokenCode == MINUS)
1059                    return true;
1060                else
1061                    return false;
1062            }
1063
1064            /// <summary>
1065            /// Checks if the next token is a MulOp token.
1066            /// </summary>
1067            /// <returns></returns>
1068            private bool isMulOp()
1069            {
1070                if (Scanner.TokenCode == MULTIPLY || Scanner.TokenCode == DIVIDE)
1071                    return true;
1072                else
1073                    return false;
1074            }
1075
1076            /// <summary>
1077            /// Checks if the next token is a Variable
1078            /// </summary>
1079            /// <returns></returns>
1080            private bool isVariable()
1081            {
1082                if (Scanner.TokenCode == IDENTIFIER)
1083                {
1084                    int index = Scanner.SymbolTable.LookupSymbol(Scanner.NextToken);
1085                    if (index != -1)
1086                    {
1087                        Symbol symbol = Scanner.SymbolTable.GetSymbol(index);
1088                        if (symbol.Kind == SymbolKind.Variable)
1089
```

```
1090                 }
1091             }
1092             return false;
1093         }
1094 
1095         /// <summary>
1096         /// Checks if the next token is an Unsigned Constant
1097         /// </summary>
1098         /// <returns></returns>
1099         private bool isUnsignedConstant()
1100         {
1101             if (Scanner.TokenCode == FLOAT || Scanner.TokenCode == INTTYPE)
1102                 return true;
1103             else
1104                 return false;
1105         }
1106 
1107         /// <summary>
1108         /// Determines if the current token could be the start of a statement.
1109         /// </summary>
1110         /// <returns>True if the token could start a statement, False if not.</returns>
1111         private bool IsStatementStart()
1112         {
1113             switch (Scanner.TokenCode)
1114             {
1115                 case IDENTIFIER:
1116                 case BEGIN:
1117                 case IF:
1118                 case WHILE:
1119                 case REPEAT:
1120                 case FOR:
1121                 case GOTO:
1122                 case WRITELN:
1123                     return true;
1124                 default:
1125                     return false;
1126             }
1127         }
1128 
```

```
1129        /// <summary>
1130        /// Determines if the current token is the start of a simple expression.
1131        /// </summary>
1132        /// <returns></returns>
1133        private bool IsSimpleExpression()
1134        {
1135            if (isSign() || isUnsignedConstant() || isVariable() || Scanner.TokenCode == LPAR)
1136                return true;
1137            else
1138                return false;
1139        }
1140
1141        /// <summary>
1142        /// Determines if the current token is a simple type keyword.
1143        /// </summary>
1144        /// <returns></returns>
1145        private bool IsSimpleType()
1146        {
1147            switch (Scanner.TokenCode)
1148            {
1149                case INTEGER:
1150                case REAL:
1151                case STRING:
1152                    return true;
1153                default:
1154                    return false;
1155            }
1156        }
1157
1158        /// <summary>
1159        /// Determines if the current token is a label.
1160        /// </summary>
1161        /// <returns></returns>
1162        private bool IsLabel()
1163        {
1164            if (Scanner.TokenCode == IDENTIFIER)
1165            {
1166                int labelIndex = Scanner.SymbolTable.LookupSymbol(Scanner.NextToken);
1167
```

```
1168                {
1169                    Symbol labelSymbol = Scanner.SymbolTable.GetSymbol(labelIndex);
1170
1171                    if (labelSymbol.Kind == SymbolKind.Label)
1172                    {
1173                        return true;
1174                    }
1175                }
1176                else
1177                {
1178                    Console.WriteLine("Error: The current token is not in the symbol table.");
1179                }
1180            }
1181            return false;
1182        }
1183
1184        #endregion
1185
1186        #region Utility Methods
1187
1188        /// <summary>
1189        /// Prints the method that is being entered or exited if TraceOn is set to true
1190        /// </summary>
1191        /// <param name="entering"></param>
1192        /// <param name="name"></param>
1193        private void Debug(bool entering, string name)
1194        {
1195            if (TraceOn)
1196            {
1197                if (entering)
1198                    Console.WriteLine("ENTERING " + name);
1199                else
1200                    Console.WriteLine("EXITING " + name);
1201            }
1202        }
1203
1204        /// <summary>
1205        /// Gets the next token and prints the token lexeme and mneumonic if Trace is on.
1206        /// </summary>
1207        private void GetNextToken()
```

```
1208        {
1209            Scanner.GetNextToken(ScannerEchoOn);
1210            if (TraceOn)
1211                Console.WriteLine("Lexeme: {0} Mnemonic: {1}", Scanner.NextToken, TokenCodes.LookupCode
                      (Scanner.TokenCode));
1212        }
1213
1214        /// <summary>
1215        /// After an error occurs this finds the begining of the next statement.
1216        /// </summary>
1217        private void Resync()
1218        {
1219            while(!IsStatementStart() && !Scanner.EndOfFile)
1220            {
1221                GetNextToken();
1222            }
1223        }
1224
1225
1226        /// <summary>
1227        /// Updates the 'kind' of the current token in the symbol table.
1228        /// </summary>
1229        private void UpdateSymbolKind(SymbolKind kind)
1230        {
1231            int tokenIndex = Scanner.SymbolTable.LookupSymbol(Scanner.NextToken);
1232            if (tokenIndex != -1)
1233            {
1234                Scanner.SymbolTable.UpdateSymbol(tokenIndex, kind, 0);
1235            }
1236            else
1237            {
1238                Console.WriteLine("Symbol not found in symbol table.");
1239            }
1240        }
1241
1242        /// <summary>
1243        /// Sets the DataType and the default value of all the provided variables in the symbol table.
1244        /// </summary>
1245        /// <param name="variables"></param>
1246        /// <param name="type"></param>
```

```
1247          private void SetVariableType(List<string> variables, string type)
1248          {
1249              int index;
1250              Symbol symbol;
1251              DataType dataType = ConvertDataType(type);
1252              int defaultIntValue = 0;
1253              double defaultRealValue = 1.1;
1254              string defaultStringValue = "string";
1255
1256              foreach (string variable in variables)
1257              {
1258                  index = Scanner.SymbolTable.LookupSymbol(variable);
1259                  if (index != -1)
1260                  {
1261                      symbol = Scanner.SymbolTable.GetSymbol(index);
1262                      if (dataType != DataType.Invalid)
1263                          symbol.DataType = dataType;
1264                      else
1265                          Console.WriteLine("Could not set value due to invalid data type.");
1266
1267                      if (dataType == DataType.Integer)
1268                          symbol.SetValue(defaultIntValue);
1269                      else if (dataType == DataType.Double)
1270                          symbol.SetValue(defaultRealValue);
1271                      else if (dataType == DataType.String)
1272                          symbol.SetValue(defaultStringValue);
1273                      else
1274                          Console.WriteLine("Could not set value dues to invalid data type.");
1275                  }
1276              }
1277          }
1278
1279          /// <summary>
1280          /// Converts a string representation of a 'type' to an instance of the enum DataType
1281          /// </summary>
1282          /// <param name="type"></param>
1283          /// <returns></returns>
1284          private DataType ConvertDataType(string type)
1285
```

```
1286            DataType dataType = DataType.Invalid;
1287
1288            switch (type.ToUpper())
1289            {
1290                case "INTEGER":
1291                    dataType = DataType.Integer;
1292                    break;
1293                case "REAL":
1294                    dataType = DataType.Double;
1295                    break;
1296                case "STRING":
1297                    dataType = DataType.String;
1298                    break;
1299                default:
1300                    Console.WriteLine("Error: Invalid Data Type.");
1301                    break;
1302            }
1303
1304            return dataType;
1305        }
1306
1307        /// <summary>
1308        /// Determines if the identifier has been previously declared as a different type.
1309        /// </summary>
1310        /// <param name="identifierType"></param>
1311        /// <returns></returns>
1312        private bool isNotPreviouslyDeclaredIdentifier(SymbolKind kind)
1313        {
1314            string declaredAs = "";
1315            string identifierType = "";
1316            List<string> identifierList = new List<string>();
1317
1318            switch (kind)
1319            {
1320                case SymbolKind.Label:
1321                    identifierList = DeclaredVariables;
1322                    declaredAs = "VARIABLE";
1323                    identifierType = "LABEL";
1324
```

```
1325                 case SymbolKind.Variable:
1326                     identifierList = DeclaredLabels;
1327                     declaredAs = "LABEL";
1328                     identifierType = "VARIABLE";
1329                     break;
1330                 default:
1331                     Console.WriteLine("Invalid Identifier Type");
1332                     return false;
1333             }
1334 
1335             if (identifierList.Contains(Scanner.NextToken))
1336             {
1337                 RedeclaredIdentifierError(identifierType, declaredAs);
1338                 return false;
1339             }
1340             else if (Scanner.NextToken == ProgramName)
1341             {
1342                 RedeclaredIdentifierError(identifierType, "ProgramName");
1343                 return false;
1344             }
1345             else
1346             {
1347                 AddToDeclaredIdentifiers(kind);
1348                 return true;
1349             }
1350         }
1351 
1352         /// <summary>
1353         /// Adds the identifier to the appropriate declated identifier list.
1354         /// </summary>
1355         /// <param name="kind"></param>
1356         private void AddToDeclaredIdentifiers(SymbolKind kind)
1357         {
1358             UpdateSymbolKind(kind);
1359             if (kind == SymbolKind.Variable)
1360             {
1361                 DeclaredVariables.Add(Scanner.NextToken);
1362             } else if (kind == SymbolKind.Label)
1363             {
```

```
1364                    DeclaredLabels.Add(Scanner.NextToken);
1365                }
1366                else
1367                    Console.WriteLine("This kind of symbol does not need to be added to declared identifiers.");
1368            }
1369
1370            #endregion
1371        }
1372    }
1373
```