Table of Contents

C:\projects\CS4100_Compiler_Design\KyleBushCompiler\KyleBushCompiler\SyntaxAnalyzer.cs 1

```
469             else if (Scanner.TokenCode == IF)
470             {
471                 GetNextToken();
472                 branchQuad = RelExpression();
473                 if (Scanner.TokenCode == THEN)
474                 {
475                     GetNextToken();
476                     Statement();
477 
478                     if (Scanner.TokenCode == ELSE)
479                     {
480                         GetNextToken();
481 
482                         patchElse = Quads.NextQuad();
483                         Quads.AddQuad(BR, -1, -1, 0);
484                         Quads.SetQuadOp3(branchQuad, Quads.NextQuad());
485 
486                         Statement();
487 
488                         Quads.SetQuadOp3(patchElse, Quads.NextQuad());
489                     }
490                     else
491                     {
492                         Quads.SetQuadOp3(branchQuad, Quads.NextQuad());
493                     }
494                 }
495                 else
496                     UnexpectedTokenError("THEN");
497             }
```

C:\projects\CS4100_Compiler_Design\KyleBushCompiler\KyleBushCompiler\SyntaxAnalyzer.cs 1

```
498             else if (Scanner.TokenCode == WHILE)
499             {
500                 GetNextToken();
501
502                 saveTop = Quads.NextQuad();
503                 branchQuad = RelExpression();
504
505                 if (Scanner.TokenCode == DO)
506                 {
507                     GetNextToken();
508                     Statement();
509                     Quads.AddQuad(BR, -1, -1, saveTop);
510                     Quads.SetQuadOp3(branchQuad, Quads.NextQuad());
511                 }
512                 else
513                     UnexpectedTokenError("DO");
514             }
```

C:\projects\CS4100_Compiler_Design\KyleBushCompiler\KyleBushCompiler\SyntaxAnalyzer.cs 1

```
530             else if (Scanner.TokenCode == FOR)
531             {
532                 GetNextToken();
533 
534                 right = Variable();
535                 if (Scanner.TokenCode == ASSIGN)
536                 {
537                     GetNextToken();
538                     left = SimpleExpression();
539                     Quads.AddQuad(MOV, left, -1, right); // Save the value of the expression in the variable.
540                     saveTop = Quads.NextQuad();
541                     if (Scanner.TokenCode == TO)
542                     {
543                         GetNextToken();
544                         limit = SimpleExpression();
545                         temp = GenSymbol();
546                         Quads.AddQuad(SUB, right, limit, temp);
547                         branchQuad = Quads.NextQuad();
548                         Quads.AddQuad(BP, temp, -1, 0);
549 
550                         if (Scanner.TokenCode == DO)
551                         {
552                             GetNextToken();
553                             Statement();
554 
555                             Quads.AddQuad(ADD, right, Plus1Index, right);
556                             Quads.AddQuad(BR, -1, -1, saveTop);
557                             Quads.SetQuadOp3(branchQuad, Quads.NextQuad());
558                         }
559                         else
560                             UnexpectedTokenError("DO");
561                     }
562                     else
563                         UnexpectedTokenError("TO");
564                 }
565                 else
566                     UnexpectedTokenError("ASSIGN");
567             }
```

```
582             else if (Scanner.TokenCode == WRITELN)
583             {
584                 GetNextToken();
585                 if (Scanner.TokenCode == LPAR)
586                 {
587                     GetNextToken();
588                     if (IsSimpleExpression())
589                     {
590                         left = SimpleExpression();
591                         Quads.AddQuad(PRINT, left, -1, -1);
592                         if (Scanner.TokenCode == RPAR)
593                             GetNextToken();
594                         else
595                             UnexpectedTokenError("RPAR");
596                     }
597                     else if (Scanner.TokenCode == IDENTIFIER)
598                     {
599                         left = Identifier();
600                         Quads.AddQuad(PRINT, left, -1, -1);
601                         if (Scanner.TokenCode == RPAR)
602                             GetNextToken();
603                         else
604                             UnexpectedTokenError("RPAR");
605                     }
606                     else if (Scanner.TokenCode == STRINGTYPE)
607                     {
608                         left = StringConst();
609                         Quads.AddQuad(PRINT, left, -1, -1);
610                         if (Scanner.TokenCode == RPAR)
611                             GetNextToken();
612                         else
613                             UnexpectedTokenError("RPAR");
614                     }
615                     else
616                         UnexpectedTokenError("SimpleExpression or IDENTIFIER or STRINGTYPE");
617                 }
618                 else
619                     UnexpectedTokenError("LPAR");
620             }
```

C:\projects\CS4100_Compiler_Design\KyleBushCompiler\KyleBushCompiler\SyntaxAnalyzer.cs 2

```
621             else
622                 UnexpectedTokenError("Statement Token");
```

C:\projects\CS4100_Compiler_Design\KyleBushCompiler\KyleBushCompiler\SyntaxAnalyzer.cs 1

```
793         /// <summary>
794         /// Implements CFG Rule: <simple expression> -> [<sign>] <term> {<addop> <term>}*
795         /// </summary>
796         /// <returns>The result of the expression.</returns>
797         private int SimpleExpression()
798         {
799             if (IsError)
800                 return -1;
801 
802             Debug(true, "SimpleExpression()");
803 
804             int left, right, temp, opcode;
805             int signVal = 0;
806 
807             if (isSign())
808             {
809                 signVal = Sign();
810             }
811 
812             left = Term();
813 
814             if (signVal == -1)
815             {
816                 Quads.AddQuad(MULTIPLY, left, Minus1Index, left);
817             }
818 
819             while (isAddOp() && !IsError)
820             {
821                 opcode = AddOp();
822                 right = Term();
823                 temp = GenSymbol();
824                 Quads.AddQuad(opcode, left, right, temp);
825                 left = temp;
826             }
827 
828             Debug(false, "SimpleExpression()");
829             return left;
830         }
831 
832         /// <summary>
```

```
833         /// Implements CFG Rule: <term> -> <factor> {<mulop> <factor> }*
834         /// </summary>
835         /// <returns></returns>
836         private int Term()
837         {
838             if (IsError)
839                 return -1;
840             int left, right, opCode, temp;
841 
842             Debug(true, "Term()");
843             left = Factor();
844 
845             while (isMulOp() && !IsError)
846             {
847                 opCode = MulOp();
848                 right = Factor();
849                 temp = GenSymbol();
850                 try
851                 {
852                     Quads.AddQuad(opCode, left, right, temp);
853                 }
854                 catch (DivideByZeroException e)
855                 {
856                     Console.WriteLine(e.Message);
857                 }
858                 left = temp;
859             }
860 
861             Debug(false, "Term()");
862             return left;
863         }
864 
865         /// <summary>
866         /// Implements CFG Rule: <factor> -> <unsigned constant> | <variable> | $LPAR <simple expression> $RPAR
867         /// </summary>
868         /// <returns></returns>
869         private int Factor()
870         {
871             if (IsError)
```

```
872                 return -1;
873
874             Debug(true, "Factor()");
875
876             int index = 0;
877
878             if (isUnsignedConstant())
879             {
880                 index = UnsignedConstant();
881             }
882             else if (isVariable())
883             {
884                 index = Variable();
885             }
886             else if (Scanner.TokenCode == LPAR)
887             {
888                 GetNextToken();
889                 index = SimpleExpression();
890                 if (Scanner.TokenCode == RPAR)
891                     GetNextToken();
892                 else
893                     UnexpectedTokenError("RPAR");
894             }
895             else
896                 UnexpectedTokenError("UNSIGNED CONSTANT or VARIABLE or LPAR");
897
898             Debug(false, "Factor()");
899             return index;
900         }
```

C:\projects\CS4100_Compiler_Design\KyleBushCompiler\KyleBushCompiler\Interpreter.cs 1

```
 88          /// <summary>
 89          /// Runs the program using the data from the given Quad Table and Symbol Table.
 90          /// Trace mode will print each quad code that the interpretter executes.
 91          /// </summary>
 92          /// <param name="quadTable">Quad Table containing all the necessary Quad Codes</param>
 93          /// <param name="symbolTable">Symbol Table containing all the necessary Symbols</param>
 94          /// <param name="TraceOn">Toggles Trace Mode on and off</param>
 95          public void InterpretQuads(QuadTable quadTable, SymbolTable symbolTable, bool TraceOn = false)
 96          {
 97              QuadTable = quadTable;
 98              SymbolTable = symbolTable;
 99              ProgramCounter = 0;
100              while (ProgramCounter < QuadTable.NextQuad())
101              {
102                  CurrentQuad = QuadTable.GetQuad(ProgramCounter);
103                  if (QuadTable.ReserveTable.isValidOpCode(CurrentQuad.OpCode))
104                  {
105                      try
106                      {
107                          switch (CurrentQuad.OpCode)
108                          {
109                              // STOP
110                              // Terminate program
111                              case STOP:
112                                  if (TraceOn)
113                                  {
114                                      PrintTrace(CurrentQuad.OpCode);
115                                  }
116                                  ProgramCounter = QuadTable.NextQuad();
117                                  break;
118                              // DIV
119                              // Compute op1 / op2, place result into op3
120                              case DIV:
121                                  if (TraceOn)
122                                  {
123                                      PrintTrace(CurrentQuad.OpCode, CurrentQuad.Op1, CurrentQuad.Op2,
                                   CurrentQuad.Op3);
124                                  }
125                                  double quotient = Convert.ToDouble(SymbolTable.GetSymbol(CurrentQuad.Op1).GetValue
```

```
                                ()) / SymbolTable.GetSymbol(CurrentQuad.Op2).GetValue();
126                                 SymbolTable.UpdateSymbol(CurrentQuad.Op3, SymbolKind.Variable, quotient);
127                                 ProgramCounter++;
128                                 break;
129                             // MUL
130                             // Compute op1 * op2, place result into op3
131                             case MUL:
132                                 if (TraceOn)
133                                 {
134                                     PrintTrace(CurrentQuad.OpCode, CurrentQuad.Op1, CurrentQuad.Op2,
                                CurrentQuad.Op3);
135                                 }
136                                 SymbolTable.UpdateSymbol(CurrentQuad.Op3, SymbolKind.Variable,
137                                     (SymbolTable.GetSymbol(CurrentQuad.Op1).GetValue() * SymbolTable.GetSymbol
                                (CurrentQuad.Op2).GetValue()));
138                                 ProgramCounter++;
139                                 break;
140                             // SUB
141                             // Compute op1 - op2, place result into op3
142                             case SUB:
143                                 if (TraceOn)
144                                 {
145                                     PrintTrace(CurrentQuad.OpCode, CurrentQuad.Op1, CurrentQuad.Op2,
                                CurrentQuad.Op3);
146                                 }
147                                 SymbolTable.UpdateSymbol(CurrentQuad.Op3, SymbolKind.Variable,
148                                     (SymbolTable.GetSymbol(CurrentQuad.Op1).GetValue() - SymbolTable.GetSymbol
                                (CurrentQuad.Op2).GetValue()));
149                                 ProgramCounter++;
150                                 break;
151                             // ADD
152                             // Compute op1 + op2, place result into op3
153                             case ADD:
154                                 if (TraceOn)
155                                 {
156                                     PrintTrace(CurrentQuad.OpCode, CurrentQuad.Op1, CurrentQuad.Op2,
                                CurrentQuad.Op3);
157                                 }
```

C:\projects\CS4100_Compiler_Design\KyleBushCompiler\KyleBushCompiler\Interpreter.cs 3

```
158                                 SymbolTable.UpdateSymbol(CurrentQuad.Op3, SymbolKind.Variable,
159                                     (SymbolTable.GetSymbol(CurrentQuad.Op1).GetValue() + SymbolTable.GetSymbol
                                    (CurrentQuad.Op2).GetValue()));
160                                 ProgramCounter++;
161                                 break;
162                             // MOV
163                             // Assign the value in op1 into op3 (op2 is ignored here)
164                             case MOV:
165                                 if (TraceOn)
166                                 {
167                                     PrintTrace(CurrentQuad.OpCode, CurrentQuad.Op1, CurrentQuad.Op3);
168                                 }
169                                 SymbolTable.UpdateSymbol(CurrentQuad.Op3, SymbolKind.Variable, SymbolTable.GetSymbol
                                    (CurrentQuad.Op1).GetValue());
170                                 ProgramCounter++;
171                                 break;
172                             // STI
173                             // Store indexed - Assign the value in op1 into op2 + offset op3
174                             case STI:
175                                 if (TraceOn)
176                                 {
177                                     PrintTrace(CurrentQuad.OpCode, CurrentQuad.Op1, CurrentQuad.Op2,
                                    CurrentQuad.Op3);
178                                 }
179                                 SymbolTable.UpdateSymbol((CurrentQuad.Op2 + CurrentQuad.Op3), SymbolKind.Variable,
                                    SymbolTable.GetSymbol(CurrentQuad.Op1).GetValue());
180                                 ProgramCounter++;
181                                 break;
182                             // LDI
183                             // Load indexed- Assign the value in op1 + offset op2, into op3
184                             case LDI:
185                                 if (TraceOn)
186                                 {
187                                     PrintTrace(CurrentQuad.OpCode, CurrentQuad.Op1, CurrentQuad.Op2,
                                    CurrentQuad.Op3);
188                                 }
189                                 SymbolTable.UpdateSymbol(CurrentQuad.Op3, SymbolKind.Variable, SymbolTable.GetSymbol
                                    (CurrentQuad.Op1 + CurrentQuad.Op2).GetValue());
```

```
190                                 ProgramCounter++;
191                                 break;
192                             // BNZ
193                             // Branch Not Zero; if op1 value <> 0, set program counter to op3
194                             case BNZ:
195                                 if (TraceOn)
196                                 {
197                                     PrintTrace(CurrentQuad.OpCode, CurrentQuad.Op3);
198                                 }
199                                 if (SymbolTable.GetSymbol(CurrentQuad.Op1).GetValue() != 0)
200                                 {
201                                     ProgramCounter = CurrentQuad.Op3;
202                                 }
203                                 else
204                                 {
205                                     ProgramCounter++;
206                                 }
207                                 break;
208                             // BNP
209                             // Branch Not Positive; if op1 value <= 0, set program counter to op3
210                             case BNP:
211                                 if (TraceOn)
212                                 {
213                                     PrintTrace(CurrentQuad.OpCode, CurrentQuad.Op3);
214                                 }
215                                 if (SymbolTable.GetSymbol(CurrentQuad.Op1).GetValue() <= 0)
216                                 {
217                                     ProgramCounter = CurrentQuad.Op3;
218                                 }
219                                 else
220                                 {
221                                     ProgramCounter++;
222                                 }
223                                 break;
224                             // BNN
225                             // Branch Not Negative; if op1 value >= 0, set program counter to op3
226                             case BNN:
227                                 if (TraceOn)
228                                 {
```

```
229                                     PrintTrace(CurrentQuad.OpCode, CurrentQuad.Op3);
230                                 }
231                                 if (SymbolTable.GetSymbol(CurrentQuad.Op1).GetValue() >= 0)
232                                 {
233                                     ProgramCounter = CurrentQuad.Op3;
234                                 }
235                                 else
236                                 {
237                                     ProgramCounter++;
238                                 }
239                                 break;
240                             // BZ
241                             // Branch Zero; if op1 value = 0, set program counter to op3
242                             case BZ:
243                                 if (TraceOn)
244                                 {
245                                     PrintTrace(CurrentQuad.OpCode, CurrentQuad.Op3);
246                                 }
247                                 if (SymbolTable.GetSymbol(CurrentQuad.Op1).GetValue() == 0)
248                                 {
249                                     ProgramCounter = CurrentQuad.Op3;
250                                 }
251                                 else
252                                 {
253                                     ProgramCounter++;
254                                 }
255                                 break;
256                             // BP
257                             // Branch Positive; if op1 value > 0, set program counter to op3
258                             case BP:
259                                 if (TraceOn)
260                                 {
261                                     PrintTrace(CurrentQuad.OpCode, CurrentQuad.Op3);
262                                 }
263                                 if (SymbolTable.GetSymbol(CurrentQuad.Op1).GetValue() > 0)
264                                 {
265                                     ProgramCounter = CurrentQuad.Op3;
266                                 }
267                                 else
268                                 {
```

C:\projects\CS4100_Compiler_Design\KyleBushCompiler\KyleBushCompiler\Interpreter.cs 6

```
269                                     ProgramCounter++;
270                                 }
271                                 break;
272                             // BN
273                             // Branch Negative; if op1 value < 0, set program counter to op3
274                             case BN:
275                                 if (TraceOn)
276                                 {
277                                     PrintTrace(CurrentQuad.OpCode, CurrentQuad.Op3);
278                                 }
279                                 if (SymbolTable.GetSymbol(CurrentQuad.Op1).GetValue() < 0)
280                                 {
281                                     ProgramCounter = CurrentQuad.Op3;
282                                 }
283                                 else
284                                 {
285                                     ProgramCounter++;
286                                 }
287                                 break;
288                             // BR
289                             // Branch (unconditional); set program counter to op3
290                             case BR:
291                                 if (TraceOn)
292                                 {
293                                     PrintTrace(CurrentQuad.OpCode, CurrentQuad.Op3);
294                                 }
295                                 ProgramCounter = CurrentQuad.Op3;
296                                 break;
297                             // BINDR
298                             // Branch (unconditional); set program counter to op3 value contents (indirect)
299                             case BINDR:
300                                 if (TraceOn)
301                                 {
302                                     PrintTrace(CurrentQuad.OpCode, SymbolTable.GetSymbol(CurrentQuad.Op3).GetValue
                                 ());
303                                 }
304                                 ProgramCounter = SymbolTable.GetSymbol(CurrentQuad.Op3).GetValue();
305                                 break;
306                             // PRINT
```

C:\projects\CS4100_Compiler_Design\KyleBushCompiler\KyleBushCompiler\Interpreter.cs 7

```
307                             // Write symbol table name and value of op 1
308                             case PRINT:
309                                 if (TraceOn)
310                                 {
311                                     PrintTrace(CurrentQuad.OpCode, CurrentQuad.Op1);
312                                 }
313                                 // Console.WriteLine($"{ SymbolTable.GetSymbol(CurrentQuad.Op1).Name} =
                                     {SymbolTable.GetSymbol(CurrentQuad.Op1).GetValue()}");
314                                 Console.WriteLine($"{SymbolTable.GetSymbol(CurrentQuad.Op1).GetValue()}");
315                                 ProgramCounter++;
316                                 break;
317                             default:
318                                 Console.WriteLine($"Invalid Opcode {CurrentQuad.OpCode}");
319                                 break;
320                         }
321                     }
322                     // Catches any exception, prints the appropriate error message, and stops running the current
                       program.
323                     catch (Exception e)
324                     {
325                         Console.WriteLine("FATAL ERROR: " + e.Message + "\n");
326                         ProgramCounter = QuadTable.NextQuad();
327                     }
328                 }
329             }
330         }
```

C:\projects\CS4100_Compiler_Design\KyleBushCompiler\KyleBushCompiler\SymbolTable.cs 1

```
128         /// <summary>
129         /// Prints the utilized rows of the symbol table in neat tabular format,
130         /// showing only the value field which is active for that row
131         /// </summary>
132         public void PrintSymbolTable()
133         {
134             Console.WriteLine("\nSYMBOL TABLE");
135             DrawHorizontalBorder(TABLEWIDTH, DIVIDER_CHAR);
136             Console.WriteLine($"|{ "Index",-6 }|{ "Name",-40 }|{ "Kind",10 }|{ "DataType",10 }|{ "Value",40 }|");
137             DrawHorizontalBorder(TABLEWIDTH, DIVIDER_CHAR);
138             foreach (var symbol in SymbolTableData)
139             {
140                 Console.WriteLine($"|{ SymbolTableData.IndexOf(symbol),-6 }|{ symbol.Name,-40 }|{ symbol.Kind,10 }|
                      { symbol.DataType,10 }|{ symbol.GetValue(),40 }|");
141             }
142             DrawHorizontalBorder(TABLEWIDTH, DIVIDER_CHAR);
143             Console.WriteLine();
144         }
```

```
318         /// <summary>
319         /// Outputs the symbol table in the following format
320         /// MOV | #TEMP1<5>| ------ | I <1>
321         /// </summary>
322         /// <param name="quadTable"></param>
323         static void QuadTableDump(QuadTable quadTable, SymbolTable symbolTable)
324         {
325             string OpCode, Op1, Op2, Op3;
326 
327             Console.WriteLine("\nQUAD TABLE");
328             DrawHorizontalBorder(TABLEWIDTH, DIVIDER_CHAR);
329             Console.WriteLine($"|{ "Opcode",-7 }|{ "Op1",44 }|{ "Op2",10 }|{ "Op3",20 }|");
330             DrawHorizontalBorder(TABLEWIDTH, DIVIDER_CHAR);
331             foreach (var quad in quadTable.QuadTableData)
332             {
333                 OpCode = quadTable.GetMnemonic(quad.OpCode);
334 
335                 if (quad.Op1 == -1)
336                 {
337                     Op1 = "------";
338                 }
339                 else
340                 {
341                     Op1 = symbolTable.GetSymbol(quad.Op1).Name + "<" + quad.Op1 + ">";
342                 }
343 
344                 if (quad.Op2 == -1)
345                 {
346                     Op2 = "------";
347                 }
348                 else
349                 {
350                     Op2 = symbolTable.GetSymbol(quad.Op2).Name + "<" + quad.Op2 + ">";
351                 }
352 
353                 if (quad.Op3 == -1)
354                 {
355                     Op3 = "------";
356                 }
357                 else if (quad.OpCode >= 8 && quad.OpCode <= 15)
```

```
358                 {
359                     Op3 = "<" + quad.Op3 + ">";
360                 }
361                 else
362                 {
363                     Op3 = symbolTable.GetSymbol(quad.Op3).Name + "<" + quad.Op3 + ">";
364                 }
365 
366                 Console.WriteLine($"|{ OpCode,-7 }|{ Op1,44 }|{ Op2,10 }|{ Op3,20 }|");
367             }
368             DrawHorizontalBorder(TABLEWIDTH, DIVIDER_CHAR);
369             Console.WriteLine();
370         }
```

C:\projects\CS4100_Compiler_Design\KyleBushCompiler\KyleBushCompiler\SyntaxAnalyzer.cs 1

```
169         /// <summary>
170         /// Implements CFG Rule: <block> -> [<label-declaration>] {<variable-dec-sec>}* <block-body>
171         /// Also contains main error handling logic.
172         /// </summary>
173         /// <returns></returns>
174         private int Block()
175         {
176             if (IsError)
177                 return -1;
178 
179             Debug(true, "Block()");
180 
181             if (Scanner.TokenCode == LABEL)
182             {
183                 Scanner.PastDeclarationSection = false;
184                 LabelDeclaration();
185             }
186 
187             while (Scanner.TokenCode == VAR && !IsError)
188             {
189                 Scanner.PastDeclarationSection = false;
190                 VariableDecSec();
191             }
192 
193             Scanner.PastDeclarationSection = true;
194 
195             BlockBody();
196 
197             // Error handling and resyncing
198             while (IsError == true && !Scanner.EndOfFile)
199             {
200                 Resync();
201                 IsError = false;
202                 PrintError = true;
203                 while (IsError == false && !Scanner.EndOfFile)
204                 {
205                     Statement();
206 
207                     if (Scanner.TokenCode == END)
208                     {
```

C:\projects\CS4100_Compiler_Design\KyleBushCompiler\KyleBushCompiler\SyntaxAnalyzer.cs 2

```
209                         GetNextToken();
210                         if (Scanner.TokenCode == PERIOD)
211                             GetNextToken();
212                         else
213                             UnexpectedTokenError("PERIOD");
214                     }
215                     else if (Scanner.TokenCode == SEMICOLON)
216                         GetNextToken();
217                     else
218                         UnexpectedTokenError("END or SEMICOLON");
219                 }
220             }
221
222             Debug(false, "Block()");
223             return -1;
224         }
```

C:\projects\CS4100_Compiler_Design\KyleBushCompiler\KyleBushCompiler\SyntaxAnalyzer.cs 1

```
1439          /// <summary>
1440          /// After an error occurs this finds the begining of the next statement.
1441          /// </summary>
1442          private void Resync()
1443          {
1444              while(!IsStatementStart() && !Scanner.EndOfFile)
1445              {
1446                  GetNextToken();
1447              }
1448          }
```

C:\projects\CS4100_Compiler_Design\KyleBushCompiler\KyleBushCompiler\SyntaxAnalyzer.cs 1

```
1221        /// <summary>
1222        /// Prints a warning message when an identifier is detected that was undeclared.
1223        /// </summary>
1224        private void UndeclaredWarning()
1225        {
1226            Console.WriteLine("\n********** Warning **********");
1227            Console.WriteLine("Line #{0}: {1}", Scanner.CurrentLineIndex, Scanner.CurrentLine);
1228            Console.WriteLine("WARNING: {0} undeclared.", Scanner.NextToken);
1229            Console.WriteLine("*****************************\n");
1230        }
1231
1232        /// <summary>
1233        /// Prints a warning message when an identifier is used as a different Kind than what it was declared.
1234        /// </summary>
1235        /// <param name="expected"></param>
1236        /// <param name="found"></param>
1237        private void DeclarationWarning(SymbolKind expected, SymbolKind found)
1238        {
1239            Console.WriteLine("\n********** Warning **********");
1240            Console.WriteLine("Line #{0}: {1}", Scanner.CurrentLineIndex, Scanner.CurrentLine);
1241            Console.WriteLine("WARNING: {0} declared as expected, but used as {1}.", Scanner.NextToken, expected,
                found);
1242            Console.WriteLine("*****************************\n");
1243        }
1244
1245        /// <summary>
1246        /// Displays
1247        /// </summary>
1248        /// <param name="used"></param>
1249        /// <param name="declared"></param>
1250        private void RedeclaredIdentifierError(string used, string declared)
1251        {
1252            IsError = true;
1253            ErrorOcurred = true;
1254
1255            Console.WriteLine("\n********** Error **********");
1256            Console.WriteLine("Line #{0}: {1}", Scanner.CurrentLineIndex, Scanner.CurrentLine);
1257            Console.WriteLine("WARNING: {0} used, but {1} declared.", used, declared);
```

C:\projects\CS4100_Compiler_Design\KyleBushCompiler\KyleBushCompiler\SyntaxAnalyzer.cs 2

```
1258              Console.WriteLine("**************************\n");
1259          }
```