```java
/* Ryan Long
 * 11/24/2020
 * CS 4100 - Compilers taught by Albert Brouillette
 * Compilers Project Part 3B: Syntax Analyzer
 *
 * This program parses a file for syntax errors. A Resync method is used to parse multiple errors.
 *
 * ***************************************************
 *                NAVIGATION:                        *
 *  FlagUnusedLabels() begins on line 328            *
 *  Nonterminal Methods begin on line 814            *
 *  Resync() begins on line 818                      *
 *  Error() begins on line 882                       *
 *  Main() (including CFG comments begins on line 1550 *
 *                                                   *
 * ***************************************************
 */

import java.io.*;
import java.util.*;

//////////////
// Classes //
//////////////

// class G holds global variables with getters and setters
class G {

    // Global ReserveTable
    private static ReserveTable R = new ReserveTable();
    public static ReserveTable GetR() { return R; }
    public static void SetR(ReserveTable r) { R = r; }

    // Global Symbol Table
    private static SymbolTable S = new SymbolTable();
    public static SymbolTable GetS() { return S; }
    public static void SetS(SymbolTable s) { S = s; }

    // verbose prints token and lexeme through GetNextToken function if set to true
    private static boolean verbose;
    public static boolean getVerbose() { return verbose; }
    public static void setVerbose(boolean value) { verbose = value; }
```

```
43
44      // echoOn prints token and lexeme information
45      private static boolean echoOn;
46      public static boolean getEchoOn() { return echoOn; }
47      public static void setEchoOn(boolean value) { echoOn = value; }
48
49      // myFile points to the file to read from
50      private static File myFile;
51      public static File GetMyFile() { return myFile; }
52      public static void SetMyFile(String filename) {
53          myFile = new File(filename);
54      }
55
56      // Global scanner
57      private static Scanner scanner;
58      public static Scanner GetScanner() { return scanner; }
59      public static void SetScanner(File filename) throws FileNotFoundException {
60
61          System.out.println("Running Syntax Analyzer on " + G.GetMyFile());
62          System.out.println("------------------------------------------------------------------------");
63
64          // Display header for Lexeme, token code, etc
65          System.out.printf("%-30s %2s %10s %12s %n", "Lexeme", "Token Code", "Mnemonic", "ST Index");
66          System.out.println("------------------------------------------------------------------------");
67          scanner = new Scanner(G.GetMyFile());
68      }
69
70      // Get line
71      private static String line;
72      public static String GetLine() { return line; }
73      public static void SetLine(String str) { line = str; }
74
75      // Get line number
76      private static int lineCount = 0;
77      public static int GetLineCount() { return lineCount; }
78      public static void SetLineCount(int count) { lineCount = count; }
79
80      // tracking char indexes
81      private static int globI = 0;
82      public static int GetGI() { return globI; }
83      public static void SetGI(int index) { globI = index; }
84
```

```
85      // Global token
86      private static String token = "init";
87      public static String GetToken() { return token; }
88      public static void SetToken(String t) { token = t; }
89
90      // Global token code
91      private static int tokenCode = 0;
92      public static int GetCode() { return tokenCode; }
93      public static void SetCode(int code) { tokenCode = code; }
94
95      // prevCh tracks the previous char when iterating through a string
96      private static char prevCh = '\0';
97      public static char GetPrevCh() { return prevCh; }
98      public static void SetPrevCh(char pch) { prevCh = pch; }
99
100     // usePrev does not iterate past current char if set to true
101     private static boolean usePrev = false;
102     public static boolean GetUsePrev() { return usePrev; }
103     public static void SetUsePrev(boolean value) { usePrev = value; }
104
105     // foundLabel is true if the current token is $LABEL
106     private static boolean foundLabel = false;
107     public static boolean GetFoundLabel() { return foundLabel; }
108     public static void SetFoundLabel(boolean value) { foundLabel = value; }
109
110     // foundError is true if the compiler catches an error.
111     // This variable is used with Resync()
112     private static boolean foundError = false;
113     public static boolean GetFoundError() { return foundError; }
114     public static void SetFoundError(boolean value) { foundError = value; }
115
116     // compiledWithoutError informs the user if the program compiled with or without errors
117     private static boolean compiledWithoutError = true;
118     public static boolean GetCompWithoutError() { return compiledWithoutError; }
119     public static void SetCompWithoutError(boolean value) { compiledWithoutError = value; }
120
121     // Tells nonterminal methods if we are syncing up with proper non-terminal
122     private static boolean resyncing = false;
123     public static boolean GetResyncing() { return resyncing; }
124     public static void SetResyncing(boolean value) { resyncing = value; }
125
126     // inBlockBody is used to see if variables have been declared before entering into block body
```

```
127     private static boolean inBlockBody = false;
128     public static boolean GetInBlockBody() { return inBlockBody; }
129     public static void SetInBlockBody(boolean value) { inBlockBody = value; }
130
131     // doNotPrint is used to suppress output of invalid tokens when Resync() finds them.
132     private static boolean doNotPrint = false;
133     public static boolean GetDoNotPrint() { return doNotPrint; }
134     public static void SetDoNotPrint(boolean value) { doNotPrint = value; }
135
136     // Tracks if end of file has been reached
137     private static boolean EndOfFile = false;
138     public static boolean GetEnd() { return EndOfFile; }
139     public static void SetEnd(boolean value) { EndOfFile = value; }
140
141 } // G
142
143 // ReserveTable holds a list of reserved words for the language (case independent).
144 // As tokens are retrieved, the ReserveTable is checked to see if the token resides there.
145 // If it does, the token is assigned the corresponding token code.
146 class ReserveTable {
147
148     HashMap<Integer, String> resWord;
149     HashMap<Integer, String> otherTokens;
150     HashMap<Integer, String> mnemonics;
151
152     public ReserveTable() {
153
154         // resWords holds PL20 reserved words for the language.
155         // otherTokens holds other important tokens for the language, mostly operators
156         // mnemonics holds token codes and corresponding 4-char mnemonics
157         resWord = new HashMap<Integer, String>();
158         otherTokens = new HashMap<Integer, String>();
159         mnemonics = new HashMap<Integer, String>();
160
161          resWord.put(0, "GOTO"); resWord.put(1, "INTEGER"); resWord.put(2, "TO");
162          resWord.put(3, "DO"); resWord.put(4, "IF"); resWord.put(5, "THEN");
163          resWord.put(6, "ELSE"); resWord.put(7, "FOR"); resWord.put(8, "OF");
164          resWord.put(9, "WRITELN"); resWord.put(10, "READLN"); resWord.put(11, "BEGIN");
165          resWord.put(12, "END"); resWord.put(13, "VAR"); resWord.put(14, "WHILE");
166          resWord.put(15, "UNIT"); resWord.put(16, "LABEL"); resWord.put(17, "REPEAT");
167          resWord.put(18, "UNTIL"); resWord.put(19, "PROCEDURE"); resWord.put(20, "DOWNTO");
168          resWord.put(21, "FUNCTION"); resWord.put(22, "RETURN"); resWord.put(23, "REAL");
```

```
169         resWord.put(24, "STRING"); resWord.put(25, "ARRAY"); resWord.put(99, "UNKN");
170
171         otherTokens.put(30, "/"); otherTokens.put(31, "*"); otherTokens.put(32, "+");
172         otherTokens.put(33, "-"); otherTokens.put(34, "("); otherTokens.put(35, ")");
173         otherTokens.put(36, ";"); otherTokens.put(37, ":="); otherTokens.put(38, ">");
174         otherTokens.put(39, "<"); otherTokens.put(40, ">="); otherTokens.put(41, "<=");
175         otherTokens.put(42, "="); otherTokens.put(43, "<>"); otherTokens.put(44, ",");
176         otherTokens.put(45, "["); otherTokens.put(46, "]"); otherTokens.put(47, ":");
177         otherTokens.put(48, ".");  otherTokens.put(99, "UNKN");
178
179         mnemonics.put(0, "GOTO");  mnemonics.put(1, "INTR");  mnemonics.put(2, "TO  ");
180         mnemonics.put(3, "DO  ");  mnemonics.put(4, "IF  ");  mnemonics.put(5, "THEN");
181         mnemonics.put(6, "ELSE");  mnemonics.put(7, "FOR ");  mnemonics.put(8, "OF  ");
182         mnemonics.put(9, "WTLN");  mnemonics.put(10, "RDLN"); mnemonics.put(11, "BGIN");
183         mnemonics.put(12, "END "); mnemonics.put(13, "VAR "); mnemonics.put(14, "WHIL");
184         mnemonics.put(15, "UNIT"); mnemonics.put(16, "LABL"); mnemonics.put(17, "REPT");
185         mnemonics.put(18, "UNTL"); mnemonics.put(19, "PROC"); mnemonics.put(20, "DNTO");
186         mnemonics.put(21, "FUNC"); mnemonics.put(22, "RTRN"); mnemonics.put(23, "REAL");
187         mnemonics.put(24, "STRG"); mnemonics.put(25, "ARAY");
188
189         mnemonics.put(30, "DIV "); mnemonics.put(31, "MUL "); mnemonics.put(32, "PLUS");
190         mnemonics.put(33, "MINU"); mnemonics.put(34, "LPRN"); mnemonics.put(35, "RPRN");
191         mnemonics.put(36, "SEMI"); mnemonics.put(37, "ASGN"); mnemonics.put(38, "GTR ");
192         mnemonics.put(39, "LESS"); mnemonics.put(40, "GTRE"); mnemonics.put(41, "LESE");
193         mnemonics.put(42, "EQL "); mnemonics.put(43, "LTGT"); mnemonics.put(44, "COMA");
194         mnemonics.put(45, "LBKT"); mnemonics.put(46, "RBKT"); mnemonics.put(47, "COLN");
195         mnemonics.put(48, "PERD");
196
197         mnemonics.put(50, "IDEN"); mnemonics.put(51, "INT "); mnemonics.put(52, "FLOT");
198         mnemonics.put(53, "STRI"); mnemonics.put(99, "UNKN");
199     }
200
201     // LookupIdentifierName checks if the Identifier token exists as a reserve word in the Reserve Table.
202     // If the token exists in the table, the token code is returned.
203     // If the token does not exist in the table, token code 50 is returned to designate an Identifier.
204     public int LookupIdentifierName(String name) {
205         Set<Map.Entry<Integer, String>> entries = resWord.entrySet();
206         Iterator<Map.Entry<Integer, String>> itr = entries.iterator();
207         Map.Entry<Integer, String> entry = null;
208         while (itr.hasNext()) {
209             entry = itr.next();
210             if (name.equalsIgnoreCase(entry.getValue())) {
```

```
                return entry.getKey();
            }
        }
        return 50;
    }

    // LookupOtherName checks if the token exists as an 'other token' in the otherToken Reserve Table.
    // If the token does not exist in the table, token code 99 is returned to designate it as 'Unknown'.
    public int LookupOtherName(String name) {
        Set<Map.Entry<Integer, String>> entries = otherTokens.entrySet();
        Iterator<Map.Entry<Integer, String>> itr = entries.iterator();
        Map.Entry<Integer, String> entry = null;
        while (itr.hasNext()) {
            entry = itr.next();
            if (name.equalsIgnoreCase(entry.getValue())) {
                return entry.getKey();
            }
        }
        return 99;
    }

    // LookupMnemonic finds the corresponding mnemonic for the given token code.
    // If the token code is not found, the mnemonic 'UNKN' is returned.
    public String LookupMnemonic(int tokenCode) {
        Set<Map.Entry<Integer, String>> entries = mnemonics.entrySet();
        Iterator<Map.Entry<Integer, String>> itr = entries.iterator();
        Map.Entry<Integer, String> entry = null;
        while (itr.hasNext()) {
            entry = itr.next();
            if (tokenCode == entry.getKey()) {
                return entry.getValue();
            }
        }
        return "UNKN";
    }

    // Look up token code given a mnemonic
    public int LookupTokenCode(String name) {
        Set<Map.Entry<Integer, String>> entries = mnemonics.entrySet();
        Iterator<Map.Entry<Integer, String>> itr = entries.iterator();
        Map.Entry<Integer, String> entry = null;
        while (itr.hasNext()) {
```

```
253             entry = itr.next();
254             if (name.equalsIgnoreCase(entry.getValue())) {
255                 return entry.getKey();
256             }
257         }
258         return 99;
259     }
260 }
261 
262 // SymbolTable holds Symbol objects in an ArrayList.
263 //If a token is an identifier (but not a reserved word)
264 // or the token is a number, it is added to the Symbol Table.
265 class SymbolTable {
266 
267     ArrayList<Symbol> symbList = new ArrayList<Symbol>();
268 
269     // Constructor to form an empty SymbolTable
270     public SymbolTable() {}
271 
272     // AddSymbol adds a Symbol to the SymbolTable. One of these three Add functions
273     // is called depending on the type of Symbol's value (String, integer, or float)
274     public int AddSymbol(String symbol, String kind, int type, String value) {
275         symbList.add(new Symbol(symbol, kind, type, value));
276         return symbList.size() - 1;
277     }
278     public int AddSymbol(String symbol, String kind, int type, int value) {
279         symbList.add(new Symbol(symbol, kind, type, value));
280         return symbList.size() - 1;
281     }
282     public int AddSymbol(String symbol, String kind, int type, double value) {
283         symbList.add(new Symbol(symbol, kind, type, value));
284         return symbList.size() - 1;
285     }
286 
287     // LookupSymbol is used to output the return index of symbol table if it exists, else -1 is returned.
288     public int LookupSymbol(String symbol) {
289         for (int i = 0; i < symbList.size(); i++) {
290             if (symbList.get(i).getName().equals(symbol))
291                 return i;
292         }
293         return -1;
294     }
```

```java

    // IsLabel says if symbol is a LABEL (true) or VAR (false)
    public boolean IsLabel(String symbol) {
        for (int i = 0; i < symbList.size(); i++) {
            String uppserSymbol = symbol.toUpperCase();
            if (symbList.get(i).getName().toUpperCase().equals(uppserSymbol)) {
                if (symbList.get(i).getKind() == "Label" |
                    symbList.get(i).getKind() == "Lused") {
                    return true;
                }
            }
        }
        return false;
    }

    // InSymbolTable returns true if the token lexeme is already in the symbol table.
    // This avoids duplicate entries in the Symbol Table.
    public boolean InSymbolTable(String symbol) {
        for (int i = 0; i < symbList.size(); i++) {
            String upperSymbol = symbol.toUpperCase();
            if (symbList.get(i).getName().toUpperCase().equals(upperSymbol)) {

                if (symbList.get(i).getKind().contains("Label")) {
                    System.out.println(symbList.get(i).getName() + " is " + symbList.get(i).getKind());
                    symbList.get(i).setKind("Lused");
                }
                return true;
            }
        }
        return false;
    }

    // This function finds and flags unused labels before exiting the program
    public void FlagUnusedLabels() {
        for (int i = 0; i < symbList.size(); i++) {

            String symb = symbList.get(i).getName();

            if (symbList.get(i).getKind().contains("Label")) {
                System.out.println("\n~~~ Warning! " + symb + " is an unused label! ~~~");
            }
        }
```

```java
    }

    // PrintSymbolTable prints the Symbol Table in neat columns,
    // displaying the token's lexeme, kind, type, and value.
    public void PrintSymbolTable() {
        System.out.println("\n\n ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~");
        System.out.println("|                                 Symbol Table                                       |");
        System.out.println(" ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~");
        System.out.printf("%-30s %-12s %-11s %s %n", "Lexeme", "Kind", "Type", "Value                        |");
        System.out.println("-------------------------------------------------------------------------------------");

        // Iterate over each item in the Symbol Table and call Symbol's overloaded toString method.
        for (int i = 0; i < symbList.size(); i++) {
            System.out.println(symbList.get(i));
        }
        System.out.println("-------------------------------------------------------------------------------------");
    }
}

// Symbol class creates Symbol objects that essentially act as memory for the computer.
// Tokens belong in the Symbol Table if (a) the token is an identifier but not a reserved word,
// (b) the token is a numeric constant (integer or float).
class Symbol {
    private String name;        // lexeme (token name)
    private String kind;            // LABEL, VARIABLE or CONST
    private int type;           // int, float, or string
    private String stringValue; // value for String type
    private int intValue;       // value for int type
    private double doubleValue; // value for double type

    // Each Symbol's value can be either a String, int, or double,
    // so each possibility has a constructor.
    public Symbol(String name, String kind, int type, String value) {       // String value
        this.name = name;
        this.kind = kind;
        this.type = type;
        stringValue = value;
    }

    public Symbol(String name, String kind, int type, int value) {      // int value
        this.name = name;
        this.kind = kind;
```

```
379         this.type = type;
380         intValue = value;
381     }
382 
383     public Symbol(String name, String kind, int type, double value) {       // double value
384         this.name = name;
385         this.kind = kind;
386         this.type = type;
387         doubleValue = value;
388     }
389 
390     // Getters
391     public String getName() { return name; }
392     public String getKind() { return kind; }
393     public int getType() { return type; }
394     public String getStringValue() { return stringValue; }
395     public int getIntValue() { return intValue; }
396     public double getDoubleValue() { return doubleValue; }
397 
398     // Setters
399     public void setKind(String kind) { this.kind = kind; }
400     public void setType(int type) { this.type = type; }
401     public void setStringValue(String value) { stringValue = value; }
402     public void setIntValue(int value) { intValue = value; }
403     public void setDoubleValue(double value) { doubleValue = value; }
404 
405     @Override
406     public String toString() {
407 
408         /* Type 0 means String value (for VAR Identifier)
409          * Type 1 means int value    (for CONST int)
410          * Type 2 means doublevalue  (for CONST double)
411          * Type 3 means String value (for CONST String)
412          *
413          * Kind 0 means VAR
414          * Kind 1 means CONST
415          * Kind 2 means LABEL
416          */
417 
418         String str = new String();
419         int padding = 85;
420 
```

```
421         switch(type) {
422         case 0:
423             str = String.format("%" + -30 + "s", name);
424             str += String.format("%" + 5 + "s", kind);
425             str += String.format("%" + 14 + "s", "STRING");
426             str += String.format("%" + 8 + "s", "0");
427             str = String.format("%1$-" + padding + "s", str);
428             str += "|";
429             return str;
430         case 1:
431             str = String.format("%" + -30 + "s", name);
432             str += String.format("%" + 5 + "s", kind);
433             str += String.format("%" + 11 + "s", "INT");
434             str += "          " + intValue;
435             str = String.format("%1$-" + padding + "s", str);
436             str += "|";
437             return str;
438         case 2:
439             str = String.format("%" + -30 + "s", name);
440             str += String.format("%" + 5 + "s", kind);
441             str += String.format("%" + 13 + "s", "FLOAT");
442             str += "        " + doubleValue;
443             str = String.format("%1$-" + padding + "s", str);
444             str += "|";
445             return str;
446         case 3:
447             str = String.format("%" + -30 + "s", name);
448             str += String.format("%" + 5 + "s", kind);
449             str += String.format("%" + 14 + "s", "STRING");
450             str += "       " + stringValue;
451             str = String.format("%1$-" + padding + "s", str);
452             str += "|";
453             return str;
454         default:
455             return "DNE";
456         }
457     }
458 } // Symbol
459 
460 public class Pt_3B {
461 
462 //////////////
```

```
//Functions //
//////////////

// GetNextChar feeds chars to GetNextToken so tokens can be assembled.
// GetNext char gets Strings from GetNextLine.
// A global index is used to determine where in the line the GetNextToken function has reached.
// If the end of the current line has been reached, GetNextChar calls GetNextLine and the index is set to 0.
// If the current line still needs to be parsed for tokens, index tracks the characters.
public static char GetNextChar() {
    if (G.GetGI() == 0) {                       // Start of line
        G.SetLine(GetNextLine());

        if (G.GetLine().isEmpty()) {            // Blank line
            return '\0';
        }
        int tempI = G.GetGI();
        G.SetGI(G.GetGI() + 1);                 // Increment globI by 1
        return G.GetLine().charAt(tempI);
    }
    else if (G.GetGI() == G.GetLine().length()) {  // End of the line has been reached
        G.SetGI(0);
        return 0;
    }
    else if (G.GetGI() > 0) {                   // Current line still needs to be parsed for tokens
            int tempI = G.GetGI();
            G.SetGI(G.GetGI() + 1);
            return G.GetLine().charAt(tempI);
    }
    else {                                      // Default: continue to parse current line
        return G.GetLine().charAt(G.GetGI());
    }
}

// GetNextLine gets the next line in the file and feeds it to GetNextChar as a String.
// If echoOn is true, the line is printed before being parsed into tokens.
public static String GetNextLine() {
    if (G.GetEnd())
        return "";

    // Nicely display line contents
    G.SetLineCount(G.GetLineCount() + 1);
    if (G.GetLineCount() > 1)
```

```
505         System.out.println();
506
507     G.SetLine(G.GetScanner().nextLine());
508
509     if (G.getEchoOn()) {        // If echoOn is true, print the line
510         System.out.println("Line " + G.GetLineCount() + ": " + G.GetLine());
511     }
512
513     // End of file is reached. GetNextChar will parse the remaining String and exit
514     if (G.GetScanner().hasNext() == false)
515         G.SetEnd(true);
516     return G.GetLine();
517 }
518
519 // GetNextToken calls GetNextChar repeatedly until a token is built.
520 public static String GetNextToken() {
521
522     int maxIdenLength = 30;
523     int maxNumLength = 16;
524
525     // Create a new token, set initial value to an empty string
526     G.SetToken("");
527
528     char ch = '\0';
529
530     while (G.GetToken() != null) {
531
532         if (G.GetGI() > 0 && G.GetUsePrev() == true) {  // Use the previous token and reset boolean usePrev
533             G.SetGI(G.GetGI() - 1);                     // Decrement globI by 1
534             G.SetUsePrev(false);
535         }
536
537         ch = GetNextChar();
538
539        while (Character.isWhitespace(ch)) {         // Eat whitespace
540            ch = GetNextChar();
541        }
542        if (Character.isLetter(ch)) {                // Create an Identifier Token
543            G.SetCode(50);
544
545             while (Character.isLetterOrDigit(ch)
546                     || ch == '_'
```

```
547                 || ch == '$') {
548             G.SetToken(G.GetToken() + ch);
549             G.SetPrevCh(ch);
550             ch = GetNextChar();
551             if (!Character.isLetterOrDigit(ch)) {   //ch not letter or digit. Grab previous token and return it
552                 G.SetUsePrev(true);;
553             }
554         }
555         if (G.GetToken().length() > maxIdenLength) {
556             G.SetToken(G.GetToken().substring(0, 30));
557             System.out.println("~~~ Warning, token exceeds length " + maxIdenLength
558                                + "! The token has been truncated. ~~~");
559         }
560 
561         // If the token is a reserved word, PrintToken prints the token code and it does not go in symbol table
562         if (G.GetCode() == 50 && G.GetR().LookupIdentifierName(G.GetToken()) != 50) { // match in ReserveTable
563             G.SetCode(G.GetR().LookupIdentifierName(G.GetToken()));
564         } else {
565 
566             // If the lexeme is not in the symbol table, add it
567             if (G.GetS().InSymbolTable(G.GetToken()) == false) {
568 
569                 if (G.GetInBlockBody() == true) {
570                     System.out.println("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~");
571                     System.out.println("Warning! " + G.GetToken() + " is not in symbol table!");
572                     System.out.println("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~");
573                 }
574 
575 
576                 // Check if the kind is VAR or LABEL
577                 if (G.GetFoundLabel()) {
578                     G.GetS().AddSymbol(G.GetToken(), "Label", 0, 0);        // Add Identifier w/kind LABEL to Symbol Table
579                 } else {
580 
581                     if (G.GetCode() == G.GetR().LookupTokenCode("STRI"))
582                         G.GetS().AddSymbol(G.GetToken(), "Con", 0, 0);      // Add String constant to Symbol Table
583                     else
584                         G.GetS().AddSymbol(G.GetToken(), "Var", 0, 0);      // Add variable to Symbol Table
585 
586                 }
587             }
588         }
```

```
589             PrintToken(G.GetToken(), G.GetCode());
590             return G.GetToken();
591         }
592         else if (Character.isDigit(ch)) {       // Create a Digit Token (integer or float)
593             G.SetCode(51);
594             boolean dotFound = false;
595 
596             while (Character.isDigit(ch)) {
597                 G.SetToken(G.GetToken() + ch);;
598                 G.SetPrevCh(ch);
599                 ch = GetNextChar();
600                 if (!Character.isDigit(ch)) {
601                     G.SetUsePrev(true);
602                 }
603 
604                 if (ch == '.' && dotFound == false) {   // Create a Float token if . is found
605                     G.SetCode(52);
606                     dotFound = true;
607                     G.SetToken(G.GetToken() + ch);;
608                     ch = GetNextChar();
609                 }
610                 if (G.GetCode() == 52) {
611                     while (Character.isDigit(ch)) {
612                         G.SetToken(G.GetToken() + ch);;
613                         ch = GetNextChar();
614                     }
615                     if (ch == 'e' || ch == 'E') {
616                         G.SetToken(G.GetToken() + ch);;
617                         ch = GetNextChar();
618                         if (ch == '+' || ch == '-') {
619                             G.SetToken(G.GetToken() + ch);;
620                             ch = GetNextChar();
621                         }
622                     }
623                     while (Character.isDigit(ch)) {
624                         G.SetToken(G.GetToken() + ch);;
625                         ch = GetNextChar();
626                     }
627                     if (G.GetToken().length() > maxNumLength) { // convert float length to string
628                         G.SetToken(G.GetToken().substring(0, maxNumLength));
629                         System.out.println("~~~ Warning, token exceeds length " + maxNumLength + "! "
630                                            + "The token has been truncated. ~~~");
```

```
631                     }
632                     if (!Character.isLetterOrDigit(ch)) {
633                         G.SetUsePrev(true);                 // ch not letter or digit, grab previous char
634                     }
635
636                     double tokenToDouble = Float.parseFloat(G.GetToken()); // convert String to float value
637
638                     if (G.GetS().InSymbolTable(G.GetToken()) == false)
639                         G.GetS().AddSymbol(G.GetToken(), "Const", 2, tokenToDouble);
640                     PrintToken(G.GetToken(), G.GetCode());
641                     return G.GetToken();                                    // return float
642                 }
643             }
644             if (G.GetToken().length() > maxNumLength) {                 // truncate int if needed
645                 System.out.println("~~~ Warning, token exceeds length " + maxNumLength + "! "
646                                    + "The token has been truncated. ~~~");
647                 G.SetToken(G.GetToken().substring(0, maxNumLength));
648
649                 int tokenToInt = Integer.MAX_VALUE;                 // set integer to max value
650
651                 if (G.GetS().InSymbolTable(G.GetToken()) == false)
652                     G.GetS().AddSymbol(G.GetToken(), "Const", 1, tokenToInt);
653                 PrintToken(G.GetToken(), G.GetCode());
654                 return G.GetToken();
655             }
656
657             int tokenToInt = Integer.parseInt(G.GetToken());
658
659             if (G.GetS().InSymbolTable(G.GetToken()) == false)
660                 G.GetS().AddSymbol(G.GetToken(), "Const", 1, tokenToInt);
661             PrintToken(G.GetToken(), G.GetCode());
662             return G.GetToken();                                    // return int
663         }
664         else if (ch == '\"') {                                      // Create a String Token, stripping quotes
665             ch = GetNextChar();
666             while (ch != '\"') {
667                 G.SetToken(G.GetToken() + ch);;
668                 ch = GetNextChar();
669                 if (ch == '\"') {                                   // String: Set token code to 53
670                     G.SetCode(53);
671
672                     // Add string constant to symbol table
```

```
673                     if (G.GetS().InSymbolTable(G.GetToken()) == false) {
674                         G.GetS().AddSymbol(G.GetToken(), "Const", 3, G.GetToken());
675                     }
676                     PrintToken(G.GetToken(), G.GetCode());
677                     return G.GetToken();
678                 } else if (ch == '\0') {
679                     // If terminating " not found by new line, throw an error
680                     System.out.println("*** Error: No terminating quote found before end of string! ***");
681                     return " ";
682                 }
683             }
684         }
685         else if (ch == '\0' && G.GetEnd() == true) {
686             break;
687         }
688         else if (ch == '\0') {                      // If token is still null, continue checking
689         }
690         else if (ch == '{' || ch == '(') {          // Create Comment Token
691             if (ch == '{') {
692                 ch = GetNextChar();                 // Catch matching }
693                 while (ch != '}') {
694                     ch = GetNextChar();
695                 }
696                 ch = GetNextChar();
697                 continue;
698             }
699             else if (ch == '(') {
700                 G.SetPrevCh(ch);
701                 ch = GetNextChar();
702                 if (ch == '*') {
703                     while (ch != ')') {
704                         ch = GetNextChar();
705                         if (ch == '\0' && G.GetEnd() == true) {
706                             System.out.println("~~~ Warning: Unterminated comment before end of file! ~~~");
707                             break;
708                         }
709                     }
710                     continue;
711                 } else {                    // ( is not followed by *, not a comment
712                     G.SetToken(G.GetToken() + G.GetPrevCh());
713                     G.SetUsePrev(true);
714                     G.SetCode(G.GetR().LookupOtherName(G.GetToken()));
```

```
715                 PrintToken(G.GetToken(), G.GetCode());
716                 return G.GetToken();
717             }
718         }
719     }
720     else if (ch == '>' || ch == '<') {      // check if token is >=, <=, or ==
721             G.SetPrevCh(ch);
722             G.SetToken(G.GetToken() + ch);;
723             ch = GetNextChar();
724             if (ch == '=') {
725                 G.SetToken(G.GetToken() + ch);;
726                 G.SetCode(G.GetR().LookupOtherName(G.GetToken()));
727                 PrintToken(G.GetToken(), G.GetCode());
728                 return G.GetToken();
729             }
730             else if (G.GetPrevCh() == '<' && ch == '>') {
731                 G.SetToken(G.GetToken() + ch);;
732                 G.SetCode(G.GetR().LookupOtherName(G.GetToken()));
733                 PrintToken(G.GetToken(), G.GetCode());
734                 return G.GetToken();
735             } else {                        // > not followed by ==
736                 G.SetCode(G.GetR().LookupOtherName(G.GetToken()));
737                 G.SetUsePrev(true);
738                 PrintToken(G.GetToken(), G.GetCode());
739                 return G.GetToken();
740             }
741     }
742     else if (ch == ':') {                   // Create a : token
743         G.SetToken(G.GetToken() + ch);;
744         G.SetUsePrev(false);
745         ch = GetNextChar();
746         if (ch == '=') {                    // Create a := token
747             G.SetToken(G.GetToken() + ch);;
748             G.SetCode(G.GetR().LookupOtherName(G.GetToken()));
749             PrintToken(G.GetToken(), G.GetCode());
750             return G.GetToken();
751         }
752         else {
753             G.SetCode(G.GetR().LookupOtherName(G.GetToken()));
754             G.SetUsePrev(true);
755             PrintToken(G.GetToken(), G.GetCode());
756             return G.GetToken();
```

```
            }
        }
        else if (ch == '+' || ch == '-' || ch == '*' || ch == '/') {    // create an operator token
            G.SetToken(G.GetToken() + ch);;
            G.SetCode(G.GetR().LookupOtherName(G.GetToken()));
            PrintToken(G.GetToken(), G.GetCode());
            return G.GetToken();
        }
        else {                                  // if trailing ) is found, not a comment
            if (ch == ')') {
                G.SetToken(G.GetToken() + ch);;
                G.SetCode(G.GetR().LookupOtherName(G.GetToken()));
                PrintToken(G.GetToken(), G.GetCode());
                return G.GetToken();
            }
            G.SetToken(G.GetToken() + ch);;
            G.SetCode(G.GetR().LookupOtherName(G.GetToken()));
            PrintToken(G.GetToken(), G.GetCode());
            return G.GetToken();
        }
    }

    // End of file reached
    G.SetEnd(true);
    return G.GetToken();
}

// PrintToken prints each token's (1) lexeme, (2) token code,
// (3) The proper 4-character mnemonic from the Reserve Table,
// and (4) For identifiers and literals added to the Symbol table,
// the corresponding Symbol Table index.
public static void PrintToken(String token, int tokenCode) {

    String str = new String();
    String type = G.GetR().LookupMnemonic(tokenCode);

    // If the token is an Identifier and NOT in ReserveTable, put it in Symbol Table
    // All int or float tokens go into ReserveTable

    if (token.length() > 0 && G.getVerbose() == true && G.GetDoNotPrint() == false) {
        // Print the Token Information
        str = String.format("%" + -30 + "s", token);
```

```
799         str += String.format("%" + 3 + "s", tokenCode);
800         str += String.format("%" + 15 + "s", type);
801 
802         // See if the lexeme exists in the symbol table
803         int symbolIndex = G.GetS().LookupSymbol(G.GetToken());
804         if (symbolIndex != -1)
805             str += String.format("%" + 12 + "s", symbolIndex);
806         else
807             str += String.format("%" + 12 + "s", "-");
808         System.out.println(str);
809     }
810 }
811 
812 ////////////////////////
813 // Method_Nonterminals //
814 ////////////////////////
815 
816 // Resynch continues parsing the file after finding an error
817 // by calling GNT until a token is found that could be the start of a statement.
818 public static void Resync() {
819 
820     while (G.GetFoundError() == true) {
821         G.SetDoNotPrint(true);
822 
823         // eat tokens until a new line is reached
824         int freshLine = G.GetLineCount() + 1;
825 
826         while (G.GetLineCount() < freshLine) {
827             G.SetToken(GetNextToken());
828         }
829 
830         //  If token is the start of <statement>, set foundError to false
831         if (
832             //G.GetS().InSymbolTable(G.GetToken()) && G.GetS().IsLabel(G.GetToken()) |
833             G.GetCode() == G.GetR().LookupTokenCode("IDEN")  |   // variable
834             G.GetCode() == G.GetR().LookupTokenCode("BGIN")  | // possible block-body
835             G.GetCode() == G.GetR().LookupTokenCode("IF  ")  |
836             G.GetCode() == G.GetR().LookupTokenCode("WHIL")  |
837             G.GetCode() == G.GetR().LookupTokenCode("REPT")  |
838             G.GetCode() == G.GetR().LookupTokenCode("FOR ")  |
839             G.GetCode() == G.GetR().LookupTokenCode("GOTO")  |
840             G.GetCode() == G.GetR().LookupTokenCode("WTLN")) {
```

```
841
842             // Valid token found, resume printing and call Statement() to resume compilation
843             G.SetDoNotPrint(false);
844             G.SetFoundError(false);
845
846             System.out.println("Found a potentially valid token: " + G.GetToken());
847
848             G.SetResyncing(true);
849             Statement_Nonterm();
850
851         // else If token is not good...
852         } else {
853
854             if (G.GetToken().length() == 0) {
855                 System.out.println("\n***** [Alert!] " +  G.GetMyFile() + " has compilation errors to be fixed! *****");
856                 G.GetS().FlagUnusedLabels();
857                 G.GetS().PrintSymbolTable();
858                 System.exit(1);
859             }
860             // Found invalid token, re-looping through while statement
861         }
862
863         // Expect . to complete program
864         G.SetToken(GetNextToken());
865
866             if (G.GetCode() == G.GetR().LookupTokenCode("PERD")) {
867                 G.SetToken(GetNextToken());
868
869                 if (G.GetToken().length() == 0) {
870                     System.out.println("\n***** [Alert!] " +  G.GetMyFile() + " has compilation errors to be fixed! *****");
871
872                     G.GetS().FlagUnusedLabels();
873                     G.GetS().PrintSymbolTable();
874                     System.exit(1);
875                 }
876         } else {
877             continue;
878         }
879     } // while FoundError == true
880 }
881
882 public static void Error(String expectedToken) {
```

```
883
884     G.SetFoundError(true);
885     G.SetCompWithoutError(false);  // informs user that the program compiled with at least 1 error
886
887     System.out.println("-------------------------------------------------------------------------");
888     System.out.println("*** " + G.GetMyFile() + " has a syntax error on line " + G.GetLineCount() + " ***");
889     System.out.println(G.GetLine());
890
891     if (G.GetToken().length() == 0) {
892         System.out.println("Expected " + expectedToken + ", but end of file has been reached.");
893     } else {
894         System.out.println("Expected " + expectedToken + ", got " + G.GetToken());
895         System.out.println("Skipping ahead to next valid statement...");
896     }
897
898     Resync();
899 }
900
901 public static void Debug(boolean entering, String name) {
902     if (G.getVerbose() == true) {
903         if (entering == true)
904             System.out.println("Entering " + name);
905         else
906             System.out.println("Exiting " + name);
907     }
908 }
909
910 public static int Program_Nonterm() {
911         Debug(true, "Program");
912         if (G.GetCode() == G.GetR().LookupTokenCode("UNIT")) {
913             G.SetToken(GetNextToken());
914
915             Prog_Identifier_Nonterm();
916
917             if (G.GetCode() == G.GetR().LookupTokenCode("SEMI")) {
918                 G.SetToken(GetNextToken());
919
920                 Block_Nonterm();
921
922                 if (G.GetCode() == G.GetR().LookupTokenCode("PERD")) {
923                     G.SetToken(GetNextToken());
924
```

```
925                 } else {
926                     Error(".");
927                 }
928             } else {
929                 Error(";");
930             }
931         } else {
932             Error("UNIT");
933         }
934 
935         Debug(false, "Program");
936 
937         if (G.GetCompWithoutError() == true)
938             System.out.println("\n~~ Congratulations, " + G.GetMyFile() + " compiled without errors! ~~");
939         else {
940             System.out.println("\n***** [Alert!] " +  G.GetMyFile() + " has compilation errors to be fixed! *****");
941 
942         }
943             return -1;
944 }
945 
946 public static int Block_Nonterm() {
947 
948     Debug(true, "Block");
949 
950     if (G.GetCode() == G.GetR().LookupTokenCode("LABL")) {
951         Label_Dec_Nonterm();
952     }
953 
954     while (G.GetCode() == G.GetR().LookupTokenCode("VAR ")) {
955         Var_Dec_Sec_NonTerm();
956     }
957 
958     Block_Body_Nonterm();
959 
960     Debug(false, "Block");
961     return -1;
962 }
963 
964 public static int Block_Body_Nonterm() {
965 
966     G.SetInBlockBody(true);
```

```
967
968     Debug(true, "Block Body");
969
970     if (G.GetCode() == G.GetR().LookupTokenCode("BGIN")) {
971         G.SetToken(GetNextToken());
972
973         do {
974             if (G.GetCode() == G.GetR().LookupTokenCode("SEMI")) {
975                 G.SetToken(GetNextToken());
976             }
977             Statement_Nonterm();
978         } while (G.GetCode() == G.GetR().LookupTokenCode("SEMI"));
979
980         if (G.GetCode() == G.GetR().LookupTokenCode("END ")) {
981             G.SetToken(GetNextToken());
982         } else {
983             Error("END or ; <statement>");
984         }
985     } else {
986         Error("BEGIN");
987     }
988
989     Debug(false, "Block Body");
990     return 1;
991 }
992
993
994 public static int Label_Dec_Nonterm() {
995
996     Debug(true, "label declaration");
997
998     if (G.GetCode() == G.GetR().LookupTokenCode("LABL")) {
999         G.SetFoundLabel(true);
1000        G.SetToken(GetNextToken());
1001        G.SetFoundLabel(false);
1002
1003        if (G.GetCode() == G.GetR().LookupTokenCode("IDEN")) {
1004            G.SetToken(GetNextToken());
1005
1006            // 0+ , IDEN
1007            while (G.GetCode() == G.GetR().LookupTokenCode("COMA")) {
1008                G.SetFoundLabel(true);
```

```
1009                G.SetToken(GetNextToken());
1010
1011                if (G.GetCode() == G.GetR().LookupTokenCode("IDEN")) {
1012                    G.SetToken(GetNextToken());
1013                    G.SetFoundLabel(false);
1014                } else {
1015                    Error("identifier");
1016                }
1017            }
1018
1019            if (G.GetCode() == G.GetR().LookupTokenCode("SEMI")) {
1020                G.SetToken(GetNextToken());
1021            } else {
1022                Error(";");
1023            }
1024        } else {
1025            Error("identifier");
1026        }
1027    } else {
1028        Error("LABEL");
1029    }
1030
1031    Debug(false, "label declaration");
1032    return -1;
1033}
1034
1035public static int Prog_Identifier_Nonterm() {
1036    Debug(true, "prog-identifier");
1037    Identifier_Nonterm();
1038    Debug(false, "prog-identifier");
1039    return -1;
1040}
1041
1042public static int Var_Dec_Sec_NonTerm() {
1043    Debug(true, "variable-dec-sec");
1044    if (G.GetCode() == G.GetR().LookupTokenCode("VAR ")) {
1045        G.SetToken(GetNextToken());
1046
1047        Var_Declar_Nonterm();
1048
1049    } else {
1050        Error("VAR");
```

```
1051    }
1052    Debug(false, "variable-dec-sec");
1053    return 1;
1054}
1055
1056// 1 or more variables can be declared
1057public static int Var_Declar_Nonterm() {
1058    Debug(true, "variable-declaration");
1059
1060    // Ensure token is identifier before moving to while loop to catch errors.
1061    if (G.GetCode() == G.GetR().LookupTokenCode("IDEN")) {
1062
1063        while (G.GetCode() == G.GetR().LookupTokenCode("IDEN")) {
1064            G.SetToken(GetNextToken());
1065
1066            // 0 or more '$COMMA identifier'
1067            while (G.GetCode() == G.GetR().LookupTokenCode("COMA")) {
1068                G.SetToken(GetNextToken());
1069                Identifier_Nonterm();
1070            }
1071
1072            if (G.GetCode() == G.GetR().LookupTokenCode("COLN")) {
1073
1074                G.SetToken(GetNextToken());
1075                Type_Nonterm();
1076
1077                if (G.GetCode() == G.GetR().LookupTokenCode("SEMI")) {
1078                    G.SetToken(GetNextToken());
1079                    // if identifier is grabbed, keep looping
1080
1081                } else {
1082                    Error(";");
1083                }
1084            } else {
1085                Error(",");
1086            }
1087        }
1088    } else {
1089        Error("identifier");
1090    }
1091
1092    Debug(false, "variable-declaration");
```

```
1093    return 1;
1094}
1095
1096
1097public static int Statement_Nonterm() {
1098    Debug(true, "statement");
1099
1100    while (G.GetS().InSymbolTable(G.GetToken()) && G.GetS().IsLabel(G.GetToken())) {
1101
1102        // In while loop, is a label
1103        G.SetToken(GetNextToken());
1104
1105        if (G.GetCode() == G.GetR().LookupTokenCode("COLN")) {
1106            G.SetToken(GetNextToken());
1107        } else {
1108            Error(":");
1109        }
1110    }
1111
1112    // At least one of the following:
1113    // <variable> $ASSIGN (<simple expression> | <string constant>)
1114    if (G.GetCode() == G.GetR().LookupTokenCode("IDEN")) {
1115        Variable_Nonterm();
1116
1117        // Exactly one of the following:
1118        if (G.GetCode() == G.GetR().LookupTokenCode("ASGN")) {
1119            G.SetToken(GetNextToken());
1120
1121            // simple expression...
1122            if (G.GetCode() == G.GetR().LookupTokenCode("PLUS") |
1123                G.GetCode() == G.GetR().LookupTokenCode("MINU") |
1124                G.GetCode() == G.GetR().LookupTokenCode("INT ") |
1125                G.GetCode() == G.GetR().LookupTokenCode("FLOT") |
1126                G.GetCode() == G.GetR().LookupTokenCode("IDEN") |
1127                G.GetCode() == G.GetR().LookupTokenCode("LPRN")) {
1128
1129                Simple_Exp_Nonterm();
1130
1131            // ...or string constant
1132            } else if (G.GetCode() == G.GetR().LookupTokenCode("STRI")) {
1133                String_Const_Nonterm();
1134            }
```

```
1135
1136            else {
1137                Error("simple expression or string constant");
1138            }
1139        } else {
1140            Error(":=");
1141        }
1142
1143    // block-body
1144    } else if (G.GetCode() == G.GetR().LookupTokenCode("BGIN")) {
1145        Block_Body_Nonterm();
1146    }
1147
1148    // IF THEN [ELSE]
1149    else if (G.GetCode() == G.GetR().LookupTokenCode("IF  ")) {
1150        G.SetToken(GetNextToken());
1151        Relexp_Nonterm();
1152
1153        if (G.GetCode() == G.GetR().LookupTokenCode("THEN")) {
1154            G.SetToken(GetNextToken());
1155            Statement_Nonterm();
1156
1157            if (G.GetCode() == G.GetR().LookupTokenCode("ELSE")) {
1158                G.SetToken(GetNextToken());
1159                Statement_Nonterm();
1160            }
1161        } else {
1162            Error("THEN");
1163        }
1164    }
1165
1166    // WHILE DO
1167    else if (G.GetCode() == G.GetR().LookupTokenCode("WHIL")) {
1168        G.SetToken(GetNextToken());
1169        Relexp_Nonterm();
1170
1171        if (G.GetCode() == G.GetR().LookupTokenCode("DO  ")) {
1172            G.SetToken(GetNextToken());
1173            Statement_Nonterm();
1174        } else {
1175            Error("DO");
1176        }
```

```
1177    }
1178
1179    // REPEAT UNTIL
1180    else if (G.GetCode() == G.GetR().LookupTokenCode("REPT")) {
1181        G.SetToken(GetNextToken());
1182        Statement_Nonterm();
1183
1184        if (G.GetCode() == G.GetR().LookupTokenCode("UNTL")) {
1185            G.SetToken(GetNextToken());
1186            Relexp_Nonterm();
1187        } else {
1188            Error("UNTIL");
1189        }
1190    }
1191
1192    // FOR ASSIGN TO DO
1193    else if (G.GetCode() == G.GetR().LookupTokenCode("FOR ")) {
1194        G.SetToken(GetNextToken());
1195        Variable_Nonterm();
1196
1197        if (G.GetCode() == G.GetR().LookupTokenCode("ASGN")) {
1198            G.SetToken(GetNextToken());
1199            Simple_Exp_Nonterm();
1200
1201            if (G.GetCode() == G.GetR().LookupTokenCode("TO  ")) {
1202                G.SetToken(GetNextToken());
1203                Simple_Exp_Nonterm();
1204
1205                if (G.GetCode() == G.GetR().LookupTokenCode("DO  ")) {
1206                    G.SetToken(GetNextToken());
1207                    Statement_Nonterm();
1208                } else {
1209                    Error("DO");
1210                }
1211            } else {
1212                Error("TO");
1213            }
1214        } else {
1215            Error(":=");
1216        }
1217    }
1218
```

```
1219    // GOTO
1220    else if (G.GetCode() == G.GetR().LookupTokenCode("GOTO")) {
1221        G.SetToken(GetNextToken());
1222        Label_Nonterm();
1223    }
1224
1225    // WRITELN
1226    else if (G.GetCode() == G.GetR().LookupTokenCode("WTLN")) {
1227        G.SetToken(GetNextToken());
1228
1229        if (G.GetCode() == G.GetR().LookupTokenCode("LPRN")) {
1230            G.SetToken(GetNextToken());
1231            //sim exp | iden| str const )
1232            if (G.GetCode() == G.GetR().LookupTokenCode("PLUS") |
1233                G.GetCode() == G.GetR().LookupTokenCode("MINU") |
1234                G.GetCode() == G.GetR().LookupTokenCode("INT ") |
1235                G.GetCode() == G.GetR().LookupTokenCode("FLOT") |
1236                G.GetCode() == G.GetR().LookupTokenCode("LPRN") |
1237                G.GetCode() == G.GetR().LookupTokenCode("IDEN")) {
1238                G.SetToken(GetNextToken());
1239            }
1240            else if (G.GetCode() == G.GetR().LookupTokenCode("IDEN")) {
1241                G.SetToken(GetNextToken());
1242            }
1243            else if (G.GetCode() == G.GetR().LookupTokenCode("STRI")) {
1244                G.SetToken(GetNextToken());
1245            }
1246            else {
1247                Error("simple expression, identifier, or string constant");
1248            }
1249        } else {
1250            Error("(");
1251        }
1252
1253        if (G.GetCode() == G.GetR().LookupTokenCode("RPRN")) {
1254            G.SetToken(GetNextToken());
1255
1256        } else {
1257            Error(")");
1258        }
1259    }
1260    else {
```

```
1261        Error("statement");
1262    }
1263
1264    Debug(false, "statement");
1265    return -1;
1266}
1267
1268public static int Variable_Nonterm() {
1269    Debug(true, "variable");
1270    Identifier_Nonterm();
1271
1272    // optional $LBKT <simple expression> $RBKT
1273    if (G.GetCode() == G.GetR().LookupTokenCode("LBKT")) {
1274        G.SetToken(GetNextToken());
1275        Simple_Exp_Nonterm();
1276
1277        if (G.GetCode() == G.GetR().LookupTokenCode("RBKT")) {
1278            G.SetToken(GetNextToken());
1279        } else {
1280            Error("]");
1281        }
1282    }
1283
1284    Debug(false, "variable");
1285    return -1;
1286}
1287
1288public static int Label_Nonterm() {
1289    Debug(true, "label");
1290
1291    // Confirm that the identifier has been declared as type 'label'
1292    // to differentiate from a variable in Statement_Nonterm()
1293    Identifier_Nonterm();
1294
1295    Debug(false, "label");
1296    return -1;
1297}
1298public static int Relexp_Nonterm() {
1299    Debug(true, "rel expression");
1300
1301    Simple_Exp_Nonterm();
1302    Relop_Nonterm();
```

```
1303    Simple_Exp_Nonterm();
1304
1305    Debug(false, "rel expression");
1306    return 1;
1307}
1308
1309public static int Relop_Nonterm() {
1310    Debug(true, "relop");
1311    if (G.GetCode() == G.GetR().LookupTokenCode("EQL ") |
1312        G.GetCode() == G.GetR().LookupTokenCode("LESS") |
1313        G.GetCode() == G.GetR().LookupTokenCode("GTR ") |
1314        G.GetCode() == G.GetR().LookupTokenCode("LTGT") |
1315        G.GetCode() == G.GetR().LookupTokenCode("LESE") |
1316        G.GetCode() == G.GetR().LookupTokenCode("GTRE")) {
1317
1318        G.SetToken(GetNextToken());
1319
1320    } else {
1321        Error("relation operator");
1322    }
1323
1324    Debug(false, "relop");
1325    return 1;
1326}
1327public static int Simple_Exp_Nonterm() {
1328    Debug(true, "simple expression");
1329
1330    // Optional sign
1331    if (G.GetCode() == G.GetR().LookupTokenCode("PLUS") ||
1332       (G.GetCode() == G.GetR().LookupTokenCode("MINU"))) {
1333        Sign_Nonterm();
1334    }
1335
1336    Term_Nonterm();
1337
1338    while (G.GetCode() == G.GetR().LookupTokenCode("PLUS") ||
1339          (G.GetCode() == G.GetR().LookupTokenCode("MINU"))) {
1340
1341        Addop_Nonterm();
1342
1343        Term_Nonterm();
1344    }
```

```
1345
1346    Debug(false, "simple expression");
1347    return -1;
1348}
1349
1350public static int Addop_Nonterm() {
1351    Debug(true, "addop");
1352    if (G.GetCode() == G.GetR().LookupTokenCode("PLUS") ||
1353       (G.GetCode() == G.GetR().LookupTokenCode("MINU"))) {
1354        G.SetToken(GetNextToken());
1355    } else {
1356        Error("+ or -");
1357    }
1358    Debug(false, "addop");
1359    return -1;
1360}
1361
1362public static int Sign_Nonterm() {
1363    Debug(true, "sign");
1364    if (G.GetCode() == G.GetR().LookupTokenCode("PLUS") ||
1365       (G.GetCode() == G.GetR().LookupTokenCode("MINU"))) {
1366        G.SetToken(GetNextToken());
1367    } else {
1368        Error("+ or -");
1369    }
1370    Debug(false, "sign");
1371    return -1;
1372}
1373
1374public static int Term_Nonterm() {
1375    Debug(true, "term");
1376
1377    Factor_Nonterm();
1378
1379    while (G.GetCode() == G.GetR().LookupTokenCode("MUL ") ||
1380          (G.GetCode() == G.GetR().LookupTokenCode("DIV "))) {
1381        Mulop_Nonterm();
1382        Factor_Nonterm();
1383    }
1384    Debug(false, "term");
1385    return -1;
1386}
```

```
1387
1388 public static int Mulop_Nonterm() {
1389     Debug(true, "mulop");
1390     if (G.GetCode() == G.GetR().LookupTokenCode("MUL ") ||
1391        (G.GetCode() == G.GetR().LookupTokenCode("DIV "))) {
1392         G.SetToken(GetNextToken());
1393     } else {
1394         Error("* or /");
1395     }
1396     Debug(false, "mulop");
1397     return -1;
1398 }
1399
1400 public static int Factor_Nonterm() {
1401     Debug(true, "Factor");
1402     if (G.GetCode() == G.GetR().LookupTokenCode("IDEN")) {
1403         Variable_Nonterm();
1404
1405     } else if (G.GetCode() == G.GetR().LookupTokenCode("FLOT") ||
1406                G.GetCode() == G.GetR().LookupTokenCode("INT ")) {
1407         Unsigned_Const_Nonterm();
1408
1409     } else if (G.GetCode() == G.GetR().LookupTokenCode("LPRN")) {
1410         G.SetToken(GetNextToken());
1411         Simple_Exp_Nonterm();
1412
1413         if (G.GetCode() == G.GetR().LookupTokenCode("RPRN")) {
1414             G.SetToken(GetNextToken());
1415         } else {
1416             Error(")");
1417         }
1418
1419     } else  {
1420         Error("float, int, or identifier");
1421     }
1422     Debug(false, "Factor");
1423     return -1;
1424 }
1425
1426 public static int Type_Nonterm() {
1427     Debug(true, "type");
1428
```

```
1429    if (G.GetCode() == G.GetR().LookupTokenCode("INTR") |
1430       (G.GetCode() == G.GetR().LookupTokenCode("REAL") |
1431       (G.GetCode() == G.GetR().LookupTokenCode("STRG")))) {
1432        Simple_Type_Nonterm();
1433
1434    } else if (G.GetCode() == G.GetR().LookupTokenCode("ARAY")) {
1435        G.SetToken(GetNextToken());
1436
1437        if (G.GetCode() == G.GetR().LookupTokenCode("LBKT")) {
1438            G.SetToken(GetNextToken());
1439
1440            if (G.GetCode() == G.GetR().LookupTokenCode("INT ")) {
1441                G.SetToken(GetNextToken());
1442
1443                if (G.GetCode() == G.GetR().LookupTokenCode("RBKT")) {
1444                    G.SetToken(GetNextToken());
1445
1446                    if (G.GetCode() == G.GetR().LookupTokenCode("OF  ")) {
1447                        G.SetToken(GetNextToken());
1448
1449                        if (G.GetCode() == G.GetR().LookupTokenCode("INT ")) {
1450                            G.SetToken(GetNextToken());
1451                        } else {
1452                            Error("integer");
1453                        }
1454                    } else {
1455                        Error("OF");
1456                    }
1457                } else {
1458                    Error("[");
1459                }
1460            } else {
1461                Error("INTTYPE");
1462            }
1463        } else {
1464            Error("[");
1465        }
1466    } else {
1467        Error("INTEGER, REAL, STRING, or ARRAY");
1468    }
1469
1470    Debug(false, "type");
```

```
1471    return -1;
1472}
1473
1474public static int Simple_Type_Nonterm() {
1475    Debug(true, "simple type");
1476    if (G.GetCode() == G.GetR().LookupTokenCode("INTR")) {
1477        G.SetToken(GetNextToken());
1478    } else if (G.GetCode() == G.GetR().LookupTokenCode("REAL")) {
1479        G.SetToken(GetNextToken());
1480    } else if (G.GetCode() == G.GetR().LookupTokenCode("STRG")) {
1481        G.SetToken(GetNextToken());
1482    } else {
1483        Error("INTEGER, FLOAT, or string");
1484    }
1485    Debug(false, "simple type");
1486    return -1;
1487}
1488
1489public static int Constant_Nonterm() {
1490    Debug(true, "constant");
1491
1492    if (G.GetCode() == G.GetR().LookupTokenCode("PLUS") |
1493       G.GetCode() == G.GetR().LookupTokenCode("MINU")) {
1494        Sign_Nonterm();
1495    }
1496
1497    Unsigned_Const_Nonterm();
1498
1499
1500    Debug(false, "constant");
1501    return -1;
1502}
1503
1504public static int Unsigned_Const_Nonterm() {
1505    Debug(true, "Unsigned Constant");
1506    Unsigned_Num_Nonterm();
1507    Debug(false, "Unsigned Constant");
1508    return -1;
1509}
1510
1511public static int Unsigned_Num_Nonterm() {
1512    Debug(true, "Unsigned Number");
```

```
1513
1514    if (G.GetCode() == G.GetR().LookupTokenCode("FLOT") ||
1515        G.GetCode() == G.GetR().LookupTokenCode("INT ")) {
1516        G.SetToken(GetNextToken());
1517    } else {
1518        Error("float or int");
1519    }
1520    Debug(false, "Unsigned Number");
1521    return -1;
1522}
1523
1524public static int Identifier_Nonterm() {
1525    Debug(true, "identifier");
1526
1527    if (G.GetCode() == G.GetR().LookupTokenCode("IDEN")) {
1528        G.SetToken(GetNextToken());
1529    } else {
1530        Error("identifier");
1531    }
1532    Debug(false, "identifier");
1533    return -1;
1534}
1535
1536public static int String_Const_Nonterm() {
1537    Debug(true, "String Constant");
1538
1539    if (G.GetCode() == G.GetR().LookupTokenCode("STRI")) {
1540        G.SetToken(GetNextToken());
1541    } else {
1542        Error("string type");
1543    }
1544
1545    Debug(false, "String Constant");
1546    return -1;
1547}
1548
1549//////////
1550// Main //
1551//////////
1552/*
1553 *  CFG Conventions:
1554 *  1) Anything prefaced by $ is a terminal token (symbol or reserved)
```

```
1555 *     Anything inside of <> is a non-terminal token
1556 *  2) An item enclosed in [] is optional unless a + follows (meaning exactly 1).
1557 *  3) An item enclosed in {} is repeatable, where * is 0 or more times, + is 1 or more times
1558 *  4) | represents OR connectors
1559 *  5) All named elements of form $FOO are token codes for terminals,
1560 *     which are defined for this language and returned by the lexical analyzer
1561 *
1562 *  CFG Syntax:
1563 *  <program> -> $UNIT <identifier> $SEMICOLON <block> $PERIOD
1564 *  where 'program' must have a unique identifier which cannot be repeated
1565 *
1566 *  <block> -> [<label-declaration>] {variable-dec-sec}* <block-body>
1567 *
1568 *  <block-body> -> $BEGIN <statement> {$SCOLN <statement>} $END
1569 *
1570 *  <label-declaration> -> $LABEL <identifier> {$COMMA <identifier>}* $SEMICOLON
1571 *
1572 *  <variable-dec-sec> -> $VAR <variable-delcaration>
1573 *
1574 *  <variable-declaration> -> {<identifier> {$COMMA <identifier>}* $COLN <type> $SEMI}+
1575 *
1576 *  <prog-identifier> -> <identifier>
1577 *
1578 *  Statements: Each statement may be preceded by one or more labels,
1579 *  each with a colon following.
1580 *  Note that exactly ONE statement optional item must appear when a <statement>
1581 *  is expected. The multi-line statement <block-body> [a BEGIN-END grouping] is one of these possible options.
1582 *  FOR loop expressions always truncate integers, and the loop always increments by 1.
1583 *  <statement> -> {<label> $COLON]}*
1584 *  [
1585 *  <variable $ASSIGN
1586 *              (<simple expression> | <string literal>) |
1587 *              <block-body> |
1588 *              $IF <relexpression> $THEN <statement> [$ELSE <statement>] |
1589 *              $WHILE <relexpression> $DO <statement> |
1590 *              $REPEAT <statement> $UNTIL <relexpression> |
1591 *              $FOR <variable> $ASSIGN <simple expression> $TO
1592 *                  <simple expression> $DO <statement> |
1593 *              $GOTO <label> |
1594 *              $WRITELN $LPAR (<simple expression> | <identifier> | stringconst>) $RPAR
1595 *  ]+
1596 *
```

```
1597 *  <variable> -> <identifier> [$LBRAK <simple expression> $RBRAK]
1598 *
1599 *  <label> -> <identifier> (must check that the identifier has been declared
1600 *  as type 'label' in order to differentiate from a variable in <statement>
1601 *  <relexpression> -> <simple expression> <relop> <simple expression>
1602 *  <relop> -> $EQ | $LESS | $GTR | $NEQ | $LEQ | $GEQ
1603 *  <simple expression> -> [<sign>] <term> {<addop> <term>}*
1604 *  <addop> -> $PLUS | $MINUS
1605 *  <sign> -> $PLUS | $MINUS
1606 *  <term> -> <factor> {<mulop> <factor>}*
1607 *  <mulop> -> $MULTIPLY | $DIVIDE
1608 *  <factor> -> <unsigned constant> | <variable> | $LPAR <simple expression> $RPAR
1609 *  <type> -> <simple type> | $ARRAY $LBRAK $INTTYPE $RBRAK $OF $INTEGER
1610 *  <simple type> -> $INTEGER | $FLOAT | $STRING
1611 *  <constant> -> [<sign>] <unsigned constant>
1612 *  <unsigned constant> -> <unsigned number>
1613 *  <unsigned number> -> $FLOATTYPE | $INTTYPE (as defined for Lexical, token code 51 or 52)
1614 *  <identifier> -> $IDENTIFIER (as defined for Lexical: <letter> {<letter>|<digit>|$|_}* )
1615 *  <stringconst> -> $STRINGTYPE (token code 53)
1616 */
1617    public static void main(String[] args) throws FileNotFoundException {
1618
1619        // Set echoOn to true to display source lines
1620        G.setEchoOn(true);
1621
1622        // Set verbose to true to display token and lexeme information
1623        G.setVerbose(false);
1624
1625        // Prepare a file to be read
1626        //G.SetMyFile("Part3BGOOD-1.txt");
1627        G.SetMyFile("Part 3B- Bad Testfile 1.txt");
1628
1629        G.SetScanner(G.GetMyFile());
1630
1631        // Get the first token
1632        G.SetToken(GetNextToken());
1633
1634        // Send the token through the syntax analyzer
1635        Program_Nonterm();
1636
1637        // Close the scanner
1638        G.GetScanner().close();
```

```
1639
1640        G.GetS().FlagUnusedLabels();
1641
1642        G.GetS().PrintSymbolTable();
1643
1644    } // main
1645} // Pt_3B
```