```java
package LexicalAnalyzer;

/**
 * Lexer
 *
 * This is the Lexer for the package and provides the getsym
 * module for analyzing the string of tokens and acts
 * as the state machine of the program
 *
 * @author  Derek Trom
 * @author Elena Corpus
 * @version 1.0
 * @since   2020-09-26
 */
public class Lexer {
    private String currentLexeme = "";
    private String currentChar = "";
    private String detectedLexeme = "";
    private String detectedToken = "";
    private String programText = "";
    private int programCounter = 0;
    private int lookAheadCounter = 0;
    private int programSize = 0;
    private int lineCounter = 1;
    private int startLine = 1;
    private int state = 0;
    private boolean hasSymbols = true;
    private boolean stopSearching = false;

    /**
     * Constructor for Lexer
     * @param programText the string of the input program
     */
    public Lexer (String programText) {
        this.programText = programText;
        this.programSize = programText.length()-1;
    }

    /**
     * State machine that generates lexemes based off
     * of the language in Language.java
     * @return detectedLexeme this will be added to the Match list
     * @throws Exception Thrown if there is an unrecognized token
     */
    public String getsym() throws Exception {
        /**
         * reset the stopSearching flag, currentLexeme, and
```

```
48          * detectedLexeme to properly search for the next
    symbol.
49          */
50         stopSearching = false;
51         currentChar = "";
52         currentLexeme = "";
53         detectedLexeme = "";
54
55         /**
56          * while we need to search
57          */
58         while (!stopSearching) {
59             /**
60              *  If programCounter has reached programSize,
    we should stop
61              *  searching because we have run out of
    symbols in the programText.
62              */
63             if (programCounter >= programSize) {
64                 stopSearching();
65                 hasSymbols = false;
66                 break;
67             }
68             /**
69              * if newline
70              */
71             if (currentChar.matches(LexicalAnalyzer.
    Language.REGEX_NEWLINE)) {
72                 lineCounter++;
73                 startLine = lineCounter;
74             }
75
76             /**
77              * switch for state transitions
78              */
79             switch (state) {
80                 /**
81                  * start state
82                  */
83                 case -1:
84                     throw new Exception("Could not compile.
    \n"+
85                             "Unrecognized token: "+
    currentChar+
86                             "\nFound on line: "+lineCounter
    );
87
88                 case LexicalAnalyzer.Language.ST_START :
89                     /**
90                      * Set the lookAheadCounter = to the
```

```
 90 programCounter so we
 91                      * know where to start looking ahead
    from if necessary.
 92                      */
 93                     lookAheadCounter = programCounter;
 94                     /**
 95                      * Set the currentChar to the char in
    the programText
 96                      * currently pointed to by the
    programCounter
 97                      */
 98                     currentChar = Character.toString(
    programText.charAt(programCounter));
 99                     /**
100                      * then, we increment the program
    counter to look at
101                      * the next character in the
    programText next time around
102                      */
103                     programCounter++;
104
105                     if (currentChar.matches(
    LexicalAnalyzer.Language.REGEX_RS_COLON)) {
106                         state = LexicalAnalyzer.Language.
    ST_COLON;
107                     } else if (currentChar.matches(
    LexicalAnalyzer.Language.REGEX_RS_LCRLYBRACK)) {
108                         state = LexicalAnalyzer.Language.
    ST_LCRLYBRK;
109                     } else if (currentChar.matches(
    LexicalAnalyzer.Language.REGEX_RS_LPAREN)) {
110                         state = LexicalAnalyzer.Language.
    ST_LPAREN;
111                     } else if (currentChar.matches(
    LexicalAnalyzer.Language.REGEX_RS_RPAREN)) {
112                         state = LexicalAnalyzer.Language.
    ST_RPAREN;
113                     } else if (currentChar.matches(
    LexicalAnalyzer.Language.REGEX_LETTER)) {
114                         state = LexicalAnalyzer.Language.
    ST_LETTER;
115                     } else if (currentChar.matches(
    LexicalAnalyzer.Language.REGEX_RS_COMMA)) {
116                         state = LexicalAnalyzer.Language.
    ST_COMMA;
117                     } else if (currentChar.matches(
    LexicalAnalyzer.Language.REGEX_RS_SEMICOLON)) {
118                         state = LexicalAnalyzer.Language.
    ST_SEMICOLON;
119                     } else if (currentChar.matches(
```

```
119 LexicalAnalyzer.Language.REGEX_RS_EQU)) {
120                         state = LexicalAnalyzer.Language.
    ST_EQU;
121                     } else if (currentChar.matches(
    LexicalAnalyzer.Language.REGEX_RS_LT)) {
122                         state = LexicalAnalyzer.Language.
    ST_LT;
123                     } else if (currentChar.matches(
    LexicalAnalyzer.Language.REGEX_RS_GT)) {
124                         state = LexicalAnalyzer.Language.
    ST_GT;
125                     } else if (currentChar.matches(
    LexicalAnalyzer.Language.REGEX_RS_PERIOD)) {
126                         state = LexicalAnalyzer.Language.
    ST_PERIOD;
127                     } else if (currentChar.matches(
    LexicalAnalyzer.Language.REGEX_DIGIT)) {
128                         state = LexicalAnalyzer.Language.
    ST_DIGIT;
129                     } else if (currentChar.matches(
    LexicalAnalyzer.Language.REGEX_RS_PLUS)) {
130                         state = LexicalAnalyzer.Language.
    ST_PLUS;
131                     } else if (currentChar.matches(
    LexicalAnalyzer.Language.REGEX_RS_MINUS)) {
132                         state = LexicalAnalyzer.Language.
    ST_MINUS;
133                     } else if (currentChar.matches(
    LexicalAnalyzer.Language.REGEX_RS_MULT)) {
134                         state = LexicalAnalyzer.Language.
    ST_MULT;
135                     } else if (currentChar.matches(
    LexicalAnalyzer.Language.REGEX_RS_DIVIDE)) {
136                         state = LexicalAnalyzer.Language.
    ST_DIVIDE;
137                     } else if (currentChar.matches(
    LexicalAnalyzer.Language.REGEX_RS_LSQBRACKET)) {
138                         state = LexicalAnalyzer.Language.
    ST_LSQBRACKET;
139                     } else if (currentChar.matches(
    LexicalAnalyzer.Language.REGEX_RS_RSQBRACKET)) {
140                         state = LexicalAnalyzer.Language.
    ST_RSQBRACKET;
141                     } else if (currentChar.matches(
    LexicalAnalyzer.Language.REGEX_SINGLEQT)) {
142                         state = LexicalAnalyzer.Language.
    ST_SINGLEQT;
143                     }else if (currentChar.matches(
    LexicalAnalyzer.Language.REGEX_WHITESPACE)) {
144                         //ignore
```

```
145                     } else {
146                         throw new Exception("Could not
    compile.\n"+
147                                 "Unrecognized token: "+
    currentChar+
148                                 "\nFound on line: "+
    lineCounter);
149                     }
150                     break;
151                 /**
152                  * state start and colon
153                  */
154                 case LexicalAnalyzer.Language.ST_COLON :
155                     /**
156                      * case is colon
157                      */
158 
159                     detectedToken = LexicalAnalyzer.
    Language.TOK_RS_COLON;
160                     currentLexeme += currentChar;
161                     detectedLexeme = currentLexeme;
162                     state = LexicalAnalyzer.Language.
    ST_COLON_EQUALS;
163                     break;
164 
165                 /**
166                  * := case
167                  */
168                 case LexicalAnalyzer.Language.
    ST_COLON_EQUALS :
169                     lookAheadCounter++;
170                     currentLexeme += Character.toString(
    programText.charAt(lookAheadCounter));
171                     if (currentLexeme.matches(
    LexicalAnalyzer.Language.REGEX_RS_ASSIGN)) {
172                         detectedToken = LexicalAnalyzer.
    Language.TOK_RS_ASSIGN;
173                         detectedLexeme = currentLexeme;
174                         lookAheadCounter++;
175                         programCounter = lookAheadCounter;
176 
177                         /**
178                          * reset
179                          */
180                         resetStateAndStopSearching();
181                         break;
182                     } else {
183                         /**
184                          * reset
185                          */
```

```
186                             resetStateAndStopSearching();
187                             break;
188                         }
189                         /**
190                          * left curly {
191                          */
192                     case LexicalAnalyzer.Language.ST_LCRLYBRK
     :
193
194                         detectedToken = LexicalAnalyzer.
    Language.TOK_LP_COMMENT;
195                         currentLexeme += currentChar;
196                         detectedLexeme = currentLexeme;
197                         lookAheadCounter++;
198                         if (Character.toString(programText.
    charAt(lookAheadCounter)).equals(LexicalAnalyzer.Language.
    REGEX_NEWLINE)) {
199                             lineCounter++;
200                             currentLexeme += Character.
    toString(programText.charAt(lookAheadCounter)).replace("\n
    ","\\n");
201                         } else {
202                             currentLexeme += Character.
    toString(programText.charAt(lookAheadCounter));
203                         }
204                         state = LexicalAnalyzer.Language.
    ST_LCRLYBRK_IGNOREALL;
205                         break;
206                     /**
207                      * left curly ignore
208                      */
209                     case LexicalAnalyzer.Language.
    ST_LCRLYBRK_IGNOREALL :
210                         if (currentLexeme.matches(
    LexicalAnalyzer.Language.REGEX_PT_CRLYCOMMENT)) {
211                             detectedLexeme = currentLexeme;
212                             lookAheadCounter++;
213                             programCounter = lookAheadCounter;
214                             /**
215                              * reset
216                              */
217                             resetStateAndStopSearching();
218                             break;
219                         } else {
220                             lookAheadCounter++;
221                             /**
222                              * no right curly detected yet
223                              */
224                             if (Character.toString(programText
    .charAt(lookAheadCounter)).equals(LexicalAnalyzer.Language
```

```
224 .REGEX_NEWLINE)) {
225                             lineCounter++;
226                             currentLexeme += Character.
    toString(programText.charAt(lookAheadCounter)).replace("\n
    ","\\n");
227                         } else {
228                             currentLexeme += Character.
    toString(programText.charAt(lookAheadCounter));
229                         }
230                         break;
231                     }
232                     /**
233                      * left parentesis
234                      */
235                 case LexicalAnalyzer.Language.ST_LPAREN :
236                     detectedToken = LexicalAnalyzer.
    Language.TOK_RS_LPAREN;
237                     currentLexeme += currentChar;
238                     detectedLexeme = currentLexeme;
239                     state = LexicalAnalyzer.Language.
    ST_LBIGRAM;
240                     break;
241                 /**
242                  * lparent and left LBIGRAM
243                  */
244                 case LexicalAnalyzer.Language.ST_LBIGRAM :
245                     lookAheadCounter++;
246 
247                     currentLexeme += Character.toString(
    programText.charAt(lookAheadCounter));
248 
249                     if (currentLexeme.matches(
    LexicalAnalyzer.Language.REGEX_RS_LBIGRAM)) {
250                         detectedToken = LexicalAnalyzer.
    Language.TOK_LP_COMMENT;
251                         detectedLexeme = currentLexeme;
252                         lookAheadCounter++;
253                         currentLexeme += Character.
    toString(programText.charAt(lookAheadCounter));
254                         state = LexicalAnalyzer.Language.
    ST_LBIGRAM_IGNOREALL;
255                         break;
256                     } else {
257                         /**
258                          * reset
259                          */
260                         resetStateAndStopSearching();
261                         break;
262                     }
263                     /**
```

```
264                      * LBIGRAM_IGNOREALL
265                      */
266                 case LexicalAnalyzer.Language.
    ST_LBIGRAM_IGNOREALL :
267                     if (currentLexeme.matches(
    LexicalAnalyzer.Language.REGEX_PT_BGRMCOMMENT)) {
268                         detectedLexeme = currentLexeme;
269                         lookAheadCounter++;
270                         programCounter = lookAheadCounter;
271                         /**
272                          * reset state
273                          */
274                         resetStateAndStopSearching();
275                         break;
276                     } else {
277                         lookAheadCounter++;
278                         if (Character.toString(programText
    .charAt(lookAheadCounter)).equals(LexicalAnalyzer.Language
    .REGEX_NEWLINE)) {
279                             // System.out.println("Next
    Line Started");
280                             lineCounter++;
281                             currentLexeme += Character.
    toString(programText.charAt(lookAheadCounter)).replace("\n
    ","\\n");
282                         } else {
283                             currentLexeme += Character.
    toString(programText.charAt(lookAheadCounter));
284                         }
285                         break;
286                     }
287                     /**
288                      * hit right parenthesis
289                      */
290                 case LexicalAnalyzer.Language.ST_RPAREN :
291                     detectedToken = LexicalAnalyzer.
    Language.TOK_RS_RPAREN;
292                     currentLexeme += currentChar;
293                     detectedLexeme = currentLexeme;
294                     resetStateAndStopSearching();
295                     break;
296                 /**
297                  * start letter found
298                  */
299                 case LexicalAnalyzer.Language.ST_LETTER :
300
301                     currentLexeme += currentChar;
302                     detectedLexeme = currentLexeme;
303                     state = LexicalAnalyzer.Language.ST_ID
    ;
```

```
304                     break;
305 
306                 /**
307                  * id
308                  */
309                 case LexicalAnalyzer.Language.ST_ID :
310                     detectedToken = LexicalAnalyzer.
    Language.TOK_LP_ID;
311                     lookAheadCounter++;
312                     currentLexeme += Character.toString(
    programText.charAt(lookAheadCounter));
313 
314                     if (currentLexeme.matches(
    LexicalAnalyzer.Language.REGEX_PT_ID)) {
315                         detectedLexeme = currentLexeme;
316                         /**
317                          * Set up a flag to determine
    whether we found a reserved word.
318                          * For each case, if the
    detectedLexeme matches any regex of a
319                          * reserved word, we will set the
    detectedToken to that of the
320                          * reserved word and waive the
    flag. We will also reset the
321                          * state and stop searching
    because we have successfully
322                          * determined the symbol, which
    should be returned to the
323                          * Driver for I/O management by
    the IOModule.
324                          */
325 
326                         boolean reservedWordFound = false;
327                         switch (detectedLexeme) {
328                             case LexicalAnalyzer.Language.
    REGEX_RW_AND :
329                                 // System.out.println("
    State Reset by ST_ID - found an RW (and)");
330                                 detectedToken =
    LexicalAnalyzer.Language.TOK_RW_AND;
331                                 reservedWordFound = true;
    break;
332                             case LexicalAnalyzer.Language.
    REGEX_RW_ARRAY :
333                                 // System.out.println("
    State Reset by ST_ID - found an RW (array)");
334                                 detectedToken =
    LexicalAnalyzer.Language.TOK_RW_ARRAY;
335                                 reservedWordFound = true;
    break;
```

```
336                                 case LexicalAnalyzer.Language.
    REGEX_RW_BEGIN :
337                                     // System.out.println("
    State Reset by ST_ID - found an RW (begin)");
338                                     detectedToken =
    LexicalAnalyzer.Language.TOK_RW_BEGIN;
339                                     reservedWordFound = true;
    break;
340                                 case LexicalAnalyzer.Language.
    REGEX_RW_BOOL :
341                                     // System.out.println("
    State Reset by ST_ID - found a TYPE (bool)");
342                                     detectedToken =
    LexicalAnalyzer.Language.TOK_TYPE_BOOL;
343                                     reservedWordFound = true;
    break;
344                                 case LexicalAnalyzer.Language.
    REGEX_RW_CHAR :
345                                     // System.out.println("
    State Reset by ST_ID - found a TYPE (char)");
346                                     detectedToken =
    LexicalAnalyzer.Language.TOK_TYPE_CHAR;
347                                     reservedWordFound = true;
    break;
348                                 case LexicalAnalyzer.Language.
    REGEX_RW_DIV :
349                                     // System.out.println("
    State Reset by ST_ID - found an RW (div)");
350                                     detectedToken =
    LexicalAnalyzer.Language.TOK_RW_DIV;
351                                     reservedWordFound = true;
    break;
352                                 case LexicalAnalyzer.Language.
    REGEX_RW_DO :
353                                     // System.out.println("
    State Reset by ST_ID - found an RW (do)");
354                                     detectedToken =
    LexicalAnalyzer.Language.TOK_RW_DO;
355                                     reservedWordFound = true;
    break;
356                                 case LexicalAnalyzer.Language.
    REGEX_RW_DOWNTO :
357                                     // System.out.println("
    State Reset by ST_ID - found an RW (downto)");
358                                     detectedToken =
    LexicalAnalyzer.Language.TOK_RW_DOWNTO;
359                                     reservedWordFound = true;
    break;
360                                 case LexicalAnalyzer.Language.
    REGEX_RW_ELSE :
```

```
361                                     // System.out.println("
    State Reset by ST_ID - found an RW (else)");
362                                     detectedToken =
    LexicalAnalyzer.Language.TOK_RW_ELSE;
363                                     reservedWordFound = true;
    break;
364                                 case LexicalAnalyzer.Language.
    REGEX_RW_END :
365                                     // System.out.println("
    State Reset by ST_ID - found an RW (end)");
366                                     detectedToken =
    LexicalAnalyzer.Language.TOK_RW_END;
367                                     reservedWordFound = true;
    break;
368                                 case LexicalAnalyzer.Language.
    REGEX_FALSE :
369                                     // System.out.println("
    State Reset by ST_ID - found an BOOLLIT (false)");
370                                     detectedToken =
    LexicalAnalyzer.Language.TOK_LIT_BOOL;
371                                     reservedWordFound = true;
    break;
372                                 case LexicalAnalyzer.Language.
    REGEX_RW_FOR :
373                                     // System.out.println("
    State Reset by ST_ID - found an RW (for)");
374                                     detectedToken =
    LexicalAnalyzer.Language.TOK_RW_FOR;
375                                     reservedWordFound = true;
    break;
376                                 case LexicalAnalyzer.Language.
    REGEX_RW_FUNCTION :
377                                     // System.out.println("
    State Reset by ST_ID - found an RW (function)");
378                                     detectedToken =
    LexicalAnalyzer.Language.TOK_RW_FUNCTION;
379                                     reservedWordFound = true;
    break;
380                                 case LexicalAnalyzer.Language.
    REGEX_RW_IF :
381                                     // System.out.println("
    State Reset by ST_ID - found an RW (if)");
382                                     detectedToken =
    LexicalAnalyzer.Language.TOK_RW_IF;
383                                     reservedWordFound = true;
    break;
384                                 case LexicalAnalyzer.Language.
    REGEX_RW_INT :
385                                     // System.out.println("
    State Reset by ST_ID - found a TYPE (int)");
```

```
386                                 detectedToken =
    LexicalAnalyzer.Language.TOK_TYPE_INT;
387                                 reservedWordFound = true;
    break;
388                             case LexicalAnalyzer.Language.
    REGEX_RW_MOD :
389                                 // System.out.println("
    State Reset by ST_ID - found an RW (mod)");
390                                 detectedToken =
    LexicalAnalyzer.Language.TOK_RW_MOD;
391                                 reservedWordFound = true;
    break;
392                             case LexicalAnalyzer.Language.
    REGEX_RW_NOT :
393                                 // System.out.println("
    State Reset by ST_ID - found an RW (not)");
394                                 detectedToken =
    LexicalAnalyzer.Language.TOK_RW_NOT;
395                                 reservedWordFound = true;
    break;
396                             case LexicalAnalyzer.Language.
    REGEX_RW_OF :
397                                 // System.out.println("
    State Reset by ST_ID - found an RW (of)");
398                                 detectedToken =
    LexicalAnalyzer.Language.TOK_RW_OF;
399                                 reservedWordFound = true;
    break;
400                             case LexicalAnalyzer.Language.
    REGEX_RW_OR :
401                                 // System.out.println("
    State Reset by ST_ID - found an RW (or)");
402                                 detectedToken =
    LexicalAnalyzer.Language.TOK_RW_OR;
403                                 reservedWordFound = true;
    break;
404                             case LexicalAnalyzer.Language.
    REGEX_RW_PROCEDURE :
405                                 // System.out.println("
    State Reset by ST_ID - found an RW (procedure)");
406                                 detectedToken =
    LexicalAnalyzer.Language.TOK_RW_PROCEDURE;
407                                 reservedWordFound = true;
    break;
408                             case LexicalAnalyzer.Language.
    REGEX_RW_PROGRAM :
409                                 // System.out.println("
    State Reset by ST_ID - found an RW (program)");
410                                 detectedToken =
    LexicalAnalyzer.Language.TOK_RW_PROGRAM;
```

```
411                                 reservedWordFound = true;
    break;
412                             case LexicalAnalyzer.Language.
    REGEX_RW_REAL :
413                                 // System.out.println("
    State Reset by ST_ID - found a TYPE (real)");
414                                 detectedToken =
    LexicalAnalyzer.Language.TOK_TYPE_REAL;
415                                 reservedWordFound = true;
    break;
416                             case LexicalAnalyzer.Language.
    REGEX_RW_STR :
417                                 // System.out.println("
    State Reset by ST_ID - found a TYPE (string)");
418                                 detectedToken =
    LexicalAnalyzer.Language.TOK_TYPE_STR;
419                                 reservedWordFound = true;
    break;
420                             case LexicalAnalyzer.Language.
    REGEX_RW_THEN :
421                                 // System.out.println("
    State Reset by ST_ID - found an RW (then)");
422                                 detectedToken =
    LexicalAnalyzer.Language.TOK_RW_THEN;
423                                 reservedWordFound = true;
    break;
424                             case LexicalAnalyzer.Language.
    REGEX_RW_TO :
425                                 // System.out.println("
    State Reset by ST_ID - found an RW (to)");
426                                 detectedToken =
    LexicalAnalyzer.Language.TOK_RW_TO;
427                                 reservedWordFound = true;
    break;
428                             case LexicalAnalyzer.Language.
    REGEX_TRUE :
429                                 // System.out.println("
    State Reset by ST_ID - found an BOOLLIT (true)");
430                                 detectedToken =
    LexicalAnalyzer.Language.TOK_LIT_BOOL;
431                                 reservedWordFound = true;
    break;
432                             case LexicalAnalyzer.Language.
    REGEX_RW_VAR :
433                                 // System.out.println("
    State Reset by ST_ID - found an RW (var)");
434                                 detectedToken =
    LexicalAnalyzer.Language.TOK_RW_VAR;
435                                 reservedWordFound = true;
    break;
```

```
436                             case LexicalAnalyzer.Language.
    REGEX_RW_WHILE :
437                                 // System.out.println("
    State Reset by ST_ID - found an RW (while)");
438                                 detectedToken =
    LexicalAnalyzer.Language.TOK_RW_WHILE;
439                                 reservedWordFound = true;
    break;
440                             default:
441                                 break;
442                         }
443                         /**
444                          * If we found a reserved word, we
     need to increment the
445                          * lookAheadCounter to point to
    the next char immediately
446                          * after it and set the
    programCounter to the lookAheadCounter.
447                          * Else, we don't change the state
     so we can continue searching
448                          */
449
450                         String nextChar = Character.
    toString(programText.charAt(lookAheadCounter+1));
451                         String tempID = currentLexeme +
    nextChar;
452
453                         if (reservedWordFound && !tempID.
    matches(LexicalAnalyzer.Language.REGEX_PT_ID)) {
454                             lookAheadCounter++;
455                             programCounter =
    lookAheadCounter;
456                             resetStateAndStopSearching();
457                             break;
458                         } else {
459                             // System.out.println("No
    state change - have not reached end of ID");
460                         }
461                         break;
462                     } else {
463
464                         /**
465                          * If we didn't match the ID regex
    , but the next char is a newline,
466                          * we should increment the
    lineCounter to maintain proper line numbering.
467                          */
468                         if (Character.toString(programText
    .charAt(lookAheadCounter)).equals(LexicalAnalyzer.Language
    .REGEX_NEWLINE)) {
```

```
469                             // System.out.println("Next
    Line Started");
470                             lineCounter++;
471                         }
472                         /**
473                          * found an ID, so we should move
    the programCounter to
474                          * however far lookAheadCounter
    got successfully, then
475                          * reset the State and stop
    searching to allow this symbol
476                          * to be returned to the Driver
    and handled by IOModule.
477                          * System.out.println("State Reset
     by ST_ID - found an ID");
478                          */
479                         programCounter = lookAheadCounter;
480                         resetStateAndStopSearching();
481                         break;
482                     }
483                     /**
484                      * comma and id
485                      */
486
487                 case LexicalAnalyzer.Language.ST_COMMA :
488                     detectedToken = LexicalAnalyzer.
    Language.TOK_RS_COMMA;
489                     currentLexeme += currentChar;
490                     detectedLexeme = currentLexeme;
491                     //reset state
492                     resetStateAndStopSearching();
493                     break;
494                 /**
495                  * comma / semicolon
496                  */
497                 case LexicalAnalyzer.Language.ST_SEMICOLON
     :
498                     // System.out.println("Entering
    ST_SEMICOLON");
499                     detectedToken = LexicalAnalyzer.
    Language.TOK_RS_SEMICOLON;
500                     currentLexeme += currentChar;
501                     detectedLexeme = currentLexeme;
502                     // System.out.println("State Reset by
    ST_SEMICOLON - found a SEMICOLON");
503                     resetStateAndStopSearching();
504                     break;
505                 /**
506                  * semi-colon, equals
507                  */
```

```
508                 case LexicalAnalyzer.Language.ST_EQU :
509                     // System.out.println("Entering ST_EQU
    ");
510                     detectedToken = LexicalAnalyzer.
    Language.TOK_RS_EQU;
511                     currentLexeme += currentChar;
512                     detectedLexeme = currentLexeme;
513                     // System.out.println("State Reset by
    ST_EQU - found a EQU");
514                     resetStateAndStopSearching();
515                     break;
516                 /**
517                  * equals/lessthan
518                  */
519                 case LexicalAnalyzer.Language.ST_LT :
520                     // System.out.println("Entering ST_LT
    ");
521
522                     detectedToken = LexicalAnalyzer.
    Language.TOK_RS_LT;
523                     currentLexeme += currentChar;
524                     detectedLexeme = currentLexeme;
525                     state = LexicalAnalyzer.Language.
    ST_LT_EQU;
526                     break;
527                 /**
528                  * lessthan equals
529                  */
530                 case LexicalAnalyzer.Language.ST_LT_EQU :
531                     // System.out.println("Entering
    ST_LT_EQU");
532                     lookAheadCounter++;
533                     // System.out.println("Lookahead
    counter incremented");
534                     currentLexeme += Character.toString(
    programText.charAt(lookAheadCounter));
535                     // System.out.println(currentLexeme);
536                     if (currentLexeme.matches(
    LexicalAnalyzer.Language.REGEX_RS_LTE)) {
537                         detectedToken = LexicalAnalyzer.
    Language.TOK_RS_LTE;
538                         detectedLexeme = currentLexeme;
539                         lookAheadCounter++;
540                         programCounter = lookAheadCounter;
541
542                         // System.out.println("State Reset
     by ST_LT_EQU - found an LTE");
543                         resetStateAndStopSearching();
544                         break;
545                     } else {
```

```
546                         // System.out.println("Leaving
    ST_LT_EQU - did not find LTE");
547                         state = LexicalAnalyzer.Language.
    ST_NE;
548                         break;
549                     }
550                     /**
551                      * lt equal ne
552                      */
553                 case LexicalAnalyzer.Language.ST_NE :
554                     // System.out.println("Entering ST_NE
    ");
555 
556                     if (currentLexeme.matches(
    LexicalAnalyzer.Language.REGEX_RS_NE)) {
557                         detectedToken = LexicalAnalyzer.
    Language.TOK_RS_NE;
558                         detectedLexeme = currentLexeme;
559                         lookAheadCounter++;
560                         programCounter = lookAheadCounter;
561 
562                         // System.out.println("State Reset
     by ST_NE - found an NE");
563                         resetStateAndStopSearching();
564                     } else {
565 
566                         // System.out.println("State Reset
     by ST_NE - did not detect an NE");
567                         resetStateAndStopSearching();
568                     }
569                     break;
570                 /**
571                  * not equals greater than
572                  */
573                 case LexicalAnalyzer.Language.ST_GT :
574                     // System.out.println("Entering ST_GT
    ");
575 
576                     detectedToken = LexicalAnalyzer.
    Language.TOK_RS_GT;
577                     currentLexeme += currentChar;
578                     detectedLexeme = currentLexeme;
579                     state = LexicalAnalyzer.Language.
    ST_GT_EQU;
580                     break;
581                 /**
582                  * greater than equal
583                  */
584                 case LexicalAnalyzer.Language.ST_GT_EQU :
585                     // System.out.println("Entering
```

```
585 ST_GT_EQU");
586                     lookAheadCounter++;
587                     // System.out.println("Lookahead
    counter incremented");
588                     currentLexeme += Character.toString(
    programText.charAt(lookAheadCounter));
589                     // System.out.println(currentLexeme);
590                     if (currentLexeme.matches(
    LexicalAnalyzer.Language.REGEX_RS_GTE)) {
591                         detectedToken = LexicalAnalyzer.
    Language.TOK_RS_GTE;
592                         detectedLexeme = currentLexeme;
593                         lookAheadCounter++;
594                         programCounter = lookAheadCounter;
595 
596                         // System.out.println("State Reset
     by ST_GT_EQU - found an GTE");
597                         resetStateAndStopSearching();
598                         break;
599                     } else {
600                         // System.out.println("State Reset
     by ST_GT_EQU - did not find GTE");
601                         resetStateAndStopSearching();
602                         break;
603                     }
604                     /**
605                      * digit
606                      */
607                 case LexicalAnalyzer.Language.ST_DIGIT :
608                     int temp = lookAheadCounter;
609                     String tempString = currentLexeme;
610                     temp ++;
611                     tempString += Character.toString(
    programText.charAt(temp));
612                     if (tempString.matches(LexicalAnalyzer
    .Language.REGEX_LETTER)) {
613                         detectedToken = Language.
    TOK_LP_ERROR;
614                         currentChar = tempString;
615                         state = -1;
616 
617                     } else {
618                         detectedToken = LexicalAnalyzer.
    Language.TOK_LIT_INT;
619                         currentLexeme += currentChar;
620                         detectedLexeme = currentLexeme;
621                         state = LexicalAnalyzer.Language.
    ST_INTEGER;
622                     }
623                     break;
```

```
624
625                 /**
626                  * integer state starts
627                  */
628                 case LexicalAnalyzer.Language.ST_INTEGER :
629                     // System.out.println("Entering
    ST_INTEGER");
630                     lookAheadCounter++;
631
632                     // System.out.println("Lookahead
    counter incremented");
633                     currentLexeme += Character.toString(
    programText.charAt(lookAheadCounter));
634                     String bug2 = Character.toString(
    programText.charAt(lookAheadCounter));
635                     if (bug2.matches(Language.REGEX_LETTER
    )) {
636
637                         currentChar = bug2;
638                         detectedToken = Language.
    TOK_LP_ERROR;
639                         state = -1;
640                         break;
641                     }
642                     if (currentLexeme.matches(
    LexicalAnalyzer.Language.REGEX_LIT_INT)) {
643                         detectedLexeme = currentLexeme;
644
645                     } else {
646
647                         String checkForDecimal = Character
    .toString(programText.charAt(lookAheadCounter));
648                         // System.out.println(
    checkForDecimal);
649                         boolean decimalFound = false;
650                         switch (checkForDecimal) {
651                             case LexicalAnalyzer.Language.
    REGEX_RS_DECIMAL :
652                                 // System.out.println("
    Leaving ST_INTEGER - found a decimal (.)");
653                                 // detectedToken =
    Language.TOK_LIT_REAL;
654                                 // detectedLexeme =
    currentLexeme;
655                                 decimalFound = true;
656                                 state = LexicalAnalyzer.
    Language.ST_REAL;
657                                 break;
658
659                             default:
```

```
660                                 break;
661                             }
662                             if (decimalFound) {
663                                 programCounter =
    lookAheadCounter;
664                                 break;
665                             }
666 
667 
668                             // System.out.println("State Reset
     by ST_INTEGER - found an INTEGER");
669                             programCounter = lookAheadCounter;
670                             resetStateAndStopSearching();
671                         }
672                         break;
673                     /**
674                      * real state
675                      */
676                     case LexicalAnalyzer.Language.ST_REAL :
677                         // System.out.println("Entering
    ST_REAL");
678                         lookAheadCounter++;
679                         // System.out.println("Lookahead
    counter incremented");
680                         currentLexeme += Character.toString(
    programText.charAt(lookAheadCounter));
681                         // System.out.println("FROM REST_REAL
    : "+currentLexeme);
682                         String bug = Character.toString(
    programText.charAt(lookAheadCounter));
683                         if (bug.matches(Language.REGEX_LETTER
    )){
684                             detectedToken = Language.
    TOK_LP_ERROR;
685                             currentChar = Character.toString(
    programText.charAt(lookAheadCounter));
686                             state = -1;
687                             break;
688                         }
689                         if (currentLexeme.matches(Language.
    REGEX_LIT_REAL)) {
690                             detectedToken = Language.
    TOK_LIT_REAL;
691                             detectedLexeme = currentLexeme;
692                             // System.out.println(
    currentLexeme);
693                         } else {
694                             if (Character.toString(programText
    .charAt(lookAheadCounter)).equals(Language.REGEX_NEWLINE
    )) {
```

```
695                             // System.out.println("Next
    Line Started");
696                             lineCounter++;
697                         }
698                         if (currentLexeme.contains(
    Language.REGEX_RS_RANGE)) {
699                             resetStateAndStopSearching();
700                             break;
701                         }
702 
703                         // System.out.println("State Reset
     by ST_REAL - found a REAL");
704                             programCounter = lookAheadCounter;
705                             resetStateAndStopSearching();
706                     }
707                     break;
708                 /**
709                  * rperiod
710                  */
711                 case LexicalAnalyzer.Language.ST_PERIOD :
712                     // System.out.println("Entering
    ST_PERIOD");
713 
714                     detectedToken = LexicalAnalyzer.
    Language.TOK_RS_PERIOD;
715                     currentLexeme += currentChar;
716                     detectedLexeme = currentLexeme;
717                     lookAheadCounter++;
718                     // System.out.println("Lookahead
    counter incremented");
719                     currentLexeme += Character.toString(
    programText.charAt(lookAheadCounter));
720                     // System.out.println(currentLexeme);
721                     if (currentLexeme.matches(
    LexicalAnalyzer.Language.REGEX_RS_RANGE)) {
722                         state = LexicalAnalyzer.Language.
    ST_RANGE;
723                         break;
724                     } else {
725                         // System.out.println("State Reset
     by ST_RANGE - did not find RANGE");
726                         resetStateAndStopSearching();
727                         hasSymbols = false;
728                         break;
729 
730                     }
731                     // System.out.println(currentLexeme);
732                     /**
733                      * range ..
734                      */
```

```
735                 case LexicalAnalyzer.Language.ST_RANGE :
736                     // System.out.println("Entering
    ST_RANGE");
737                     detectedToken = LexicalAnalyzer.
    Language.TOK_RS_RANGE;
738                     detectedLexeme = currentLexeme;
739                     lookAheadCounter++;
740                     programCounter = lookAheadCounter;
741 
742                     // System.out.println("State Reset by
    ST_RANGE - found a RANGE");
743                     resetStateAndStopSearching();
744                     break;
745                 /**
746                  *plus
747                  */
748                 case LexicalAnalyzer.Language.ST_PLUS :
749                     // System.out.println("Entering
    ST_PLUS");
750                     detectedToken = LexicalAnalyzer.
    Language.TOK_RS_PLUS;
751                     currentLexeme += currentChar;
752                     detectedLexeme = currentLexeme;
753                     // System.out.println("State Reset by
    ST_PLUS - found a PLUS");
754                     resetStateAndStopSearching();
755                     break;
756                 /**
757                  * minus
758                  */
759                 case LexicalAnalyzer.Language.ST_MINUS :
760                     // System.out.println("Entering
    ST_MINUS");
761                     detectedToken = LexicalAnalyzer.
    Language.TOK_RS_MINUS;
762                     currentLexeme += currentChar;
763                     detectedLexeme = currentLexeme;
764                     // System.out.println("State Reset by
    ST_MINUS - found a MINUS");
765                     resetStateAndStopSearching();
766                     break;
767                 /**
768                  * multiply
769                  */
770                 case LexicalAnalyzer.Language.ST_MULT :
771                     // System.out.println("Entering
    ST_MULT");
772                     detectedToken = LexicalAnalyzer.
    Language.TOK_RS_MULT;
773                     currentLexeme += currentChar;
```

```
774                     detectedLexeme = currentLexeme;
775                     // System.out.println("State Reset by
    ST_MULT - found a MULT");
776                     resetStateAndStopSearching();
777                     break;
778                 /**
779                  * divide
780                  */
781                 case LexicalAnalyzer.Language.ST_DIVIDE :
782                     // System.out.println("Entering
    ST_DIVIDE");
783                     detectedToken = LexicalAnalyzer.
    Language.TOK_RW_DIV;
784                     currentLexeme += currentChar;
785                     detectedLexeme = currentLexeme;
786                     // System.out.println("State Reset by
    ST_DIVIDE - found a DIVIDE");
787                     resetStateAndStopSearching();
788                     break;
789                 /**
790                  * left square bracket
791                  */
792                 case LexicalAnalyzer.Language.
    ST_LSQBRACKET :
793                     // System.out.println("Entering
    ST_LSQBRACKET");
794                     detectedToken = LexicalAnalyzer.
    Language.TOK_RS_LSQBRACKET;
795                     currentLexeme += currentChar;
796                     detectedLexeme = currentLexeme;
797                     // System.out.println("State Reset by
    ST_LSQBRACKET - found a LSQBRACKET");
798                     resetStateAndStopSearching();
799                     break;
800                 /**
801                  * right square bracket
802                  */
803                 case LexicalAnalyzer.Language.
    ST_RSQBRACKET :
804                     // System.out.println("Entering
    ST_RSQBRACKET");
805                     detectedToken = Language.
    TOK_RS_RSQBRACKET;
806                     currentLexeme += currentChar;
807                     detectedLexeme = currentLexeme;
808                     // System.out.println("State Reset by
    ST_RSQBRACKET - found a RSQBRACKET");
809                     resetStateAndStopSearching();
810                     break;
811                 /**
```

```
812                  * Single quote
813                  */
814                 case LexicalAnalyzer.Language.ST_SINGLEQT
     :
815                     // System.out.println("Entering
    ST_SINGLEQT");
816                     detectedToken = LexicalAnalyzer.
    Language.TOK_LIT_STR;
817                     currentLexeme += currentChar;
818                     detectedLexeme = currentLexeme;
819                     lookAheadCounter++;
820                     if (Character.toString(programText.
    charAt(lookAheadCounter)).equals(Language.REGEX_NEWLINE
    )) {
821                         // System.out.println("Next Line
    Started");
822                         lineCounter++;
823                         currentLexeme += Character.
    toString(programText.charAt(lookAheadCounter)).replace("\n
    ","\\n");
824                     } else {
825                         currentLexeme += Character.
    toString(programText.charAt(lookAheadCounter));
826                     }
827                     // System.out.println("Leaving state
    ST_SINGLEQT - found a SINGLEQT");
828                     state = LexicalAnalyzer.Language.
    ST_SINGLEQT_ACCEPTALL;
829                     break;
830                 /**
831                  * single quote accept all
832                  */
833                 case LexicalAnalyzer.Language.
    ST_SINGLEQT_ACCEPTALL :
834                     // System.out.println("Entering
    ST_SINGLEQT_ACCEPTALL");
835                     if (currentLexeme.matches(
    LexicalAnalyzer.Language.REGEX_LIT_STRING)) {
836                         detectedLexeme = currentLexeme;
837                         if (detectedLexeme.replace("'",""
    ).length()==1) {
838                             detectedToken =
    LexicalAnalyzer.Language.TOK_LIT_CHAR;
839                         }
840                         lookAheadCounter++;
841                         programCounter = lookAheadCounter;
842
843                         // System.out.println("State Reset
     by ST_SINGLEQT_ACCEPTALL - found a STRING");
844                         resetStateAndStopSearching();
```

```
845                             break;
846                         } else {
847                             lookAheadCounter++;
848                             if (Character.toString(programText
    .charAt(lookAheadCounter)).equals(LexicalAnalyzer.Language
    .REGEX_NEWLINE)) {
849                                 // System.out.println("Next
    Line Started");
850                                 lineCounter++;
851                                 currentLexeme += Character.
    toString(programText.charAt(lookAheadCounter)).replace("\n
    ","\\n");
852                             } else {
853                                 currentLexeme += Character.
    toString(programText.charAt(lookAheadCounter));
854                             }
855                             break;
856                         }
857                         /**
858                          * default case for switch
859                          */
860                     default:
861                         // System.out.println("State Reset by
    default case");
862                         resetStateAndStopSearching();
863                         break;
864                 }
865             }
866 
867             return detectedLexeme;
868         }
869 
870         /**
871          * Determines if the Lexer is ready
872          * @return boolean hasSymbols
873          */
874         public boolean isReady() {
875             return hasSymbols;
876         }
877 
878         /**
879          * Gets and returns the detected token
880          * @return String detectedToken
881          */
882         public String getDetectedToken() {
883             return detectedToken;
884         }
885 
886         /**
887          * gets the lexeme
```

```
888      * @return String detectedLexeme
889      */
890     public String getDetectedLexeme() {
891         return detectedLexeme;
892     }
893 
894     /**
895      * Resets the state machine and stops the search
896      */
897     public void resetStateAndStopSearching() {
898         state = LexicalAnalyzer.Language.ST_START;
899         stopSearching();
900     }
901 
902     /**
903      * Used to reset to top of switch state machine
904      */
905     public void resetState() {
906         System.out.println("STATE RESET from resetState()"
    );
907         state = LexicalAnalyzer.Language.ST_START;
908     }
909 
910     /**
911      * Stops the program from searching
912      */
913     public void stopSearching() {
914         stopSearching = true;
915     }
916 }
917 
```