

```

1  import components.map.Map;
12
13 /**
14  * Layered implementation of secondary method {@code parse} for {@code Program}.
15  *
16  * @author Gabe Azzarita and Ty Fredrick
17  *
18  */
19 public final class Program1Parse1 extends Program1 {
20
21     /*
22      * Private members -----
23      */
24
25     /**
26      * Parses a single BL instruction from {@code tokens} returning the
27      * instruction name as the value of the function and the body of the
28      * instruction in {@code body}.
29      *
30      * @param tokens
31      *         the input tokens
32      * @param body
33      *         the instruction body
34      * @return the instruction name
35      * @replaces body
36      * @updates tokens
37      * @requires <pre>
38      *   [<"INSTRUCTION"> is a prefix of tokens] and
39      *   [<Tokenizer.END_OF_INPUT> is a suffix of tokens]
40      * </pre>
41      * @ensures <pre>
42      *   if [an instruction string is a proper prefix of #tokens] and
43      *     [the beginning name of this instruction equals its ending name] and
44      *     [the name of this instruction does not equal the name of a primitive
45      *       instruction in the BL language] then
46      *     parseInstruction = [name of instruction at start of #tokens] and
47      *     body = [Statement corresponding to the block string that is the body of
48      *       the instruction string at start of #tokens] and
49      *     #tokens = [instruction string at start of #tokens] * tokens
50      * else
51      *   [report an appropriate error message to the console and terminate client]
52      * </pre>
53      */
54     private static String parseInstruction(Queue<String> tokens,
55         Statement body) {
56         assert tokens != null : "Violation of: tokens is not null";
57         assert body != null : "Violation of: body is not null";
58         assert tokens.length() > 0 && tokens.front().equals("INSTRUCTION") : ""
59             + "Violation of: <\"INSTRUCTION\"> is proper prefix of tokens";
60
61         // check sytanx for INSTRUCTION
62         String instruction = tokens.dequeue();
63         Reporter.assertElseFatalError(instruction.equals("INSTRUCTION"),
64             "Recieved: " + instruction + ", Expected: INSTRUCTION.");
65
66         // check for proper identifier
67         String identifier = tokens.dequeue();
68         Reporter.assertElseFatalError(Tokenizer.isIdentifier(identifier),
69             identifier + ": invalid identifier.");

```

```

70
71     // check syntax for IS
72     String is = tokens.dequeue();
73     Reporter.assertElseFatalError(is.equals("IS"),
74         "Recieved: " + is + ", Expected: IS.");
75
76     // parse body
77     body.parseBlock(tokens);
78
79     // check syntax for END and identifier
80     String end = tokens.dequeue();
81     Reporter.assertElseFatalError(end.equals("END"),
82         "Recieved: " + end + " , Expected: END.");
83
84     Reporter.assertElseFatalError(tokens.dequeue().equals(identifier),
85         "Identifier at start does not match identifier at end.");
86
87     return identifier;
88 }
89
90 /*
91  * Constructors -----
92  */
93
94 /**
95  * No-argument constructor.
96  */
97 public Program1Parse1() {
98     super();
99 }
100
101 /*
102  * Public methods -----
103  */
104
105 @Override
106 public void parse(SimpleReader in) {
107     assert in != null : "Violation of: in is not null";
108     assert in.isOpen() : "Violation of: in.is_open";
109     Queue<String> tokens = Tokenizer.tokens(in);
110     this.parse(tokens);
111 }
112
113 @Override
114 public void parse(Queue<String> tokens) {
115     assert tokens != null : "Violation of: tokens is not null";
116     assert tokens.length() > 0 : ""
117         + "Violation of: Tokenizer.END_OF_INPUT is a suffix of tokens";
118
119     // check syntax for PROGRAM, identifier, and IS
120     String program = tokens.dequeue();
121     Reporter.assertElseFatalError(program.equals("PROGRAM"),
122         "Recieved: " + program + ", expected: PROGRAM.");
123
124     String identifier = tokens.dequeue();
125     Reporter.assertElseFatalError(Tokenizer.isIdentifier(identifier),
126         identifier + ": invalid identifier.");
127
128     String is = tokens.dequeue();

```

```

129     Reporter.assertElseFatalError(is.equals("IS"),
130         "Recieved: " + is + ", Expected: IS.");
131
132     /*
133     * create and fill context so long as front token is INSTRUCTION and the
134     * instruction name is not already defined in the map
135     */
136
137     Map<String, Statement> context = this.newContext();
138     if (tokens.length() > 0) {
139
140         while (tokens.front().equals("INSTRUCTION")) {
141
142             Statement block = this.newBody();
143             String instructionName = parseInstruction(tokens, block);
144
145             Reporter.assertElseFatalError(!context.containsKey(instructionName),
146                 "Cannot have duplicate user defined instructions.");
147
148             context.add(instructionName, block);
149
150         }
151     }
152
153     // check syntax
154     String begin = tokens.dequeue();
155     Reporter.assertElseFatalError(begin.equals("BEGIN"),
156         "Recieved: " + begin + ", Expected: BEGIN");
157
158     // create and parse new body
159     Statement body = this.newBody();
160     body.parseBlock(tokens);
161
162     // check syntax for END and identifier and END OF INPUT
163     String end = tokens.dequeue();
164     Reporter.assertElseFatalError(end.equals("END"),
165         "Recieved: " + end + " , Expected: END.");
166
167     Reporter.assertElseFatalError(tokens.dequeue().equals(identifier),
168         "Identifier at start does not match identifier at end.");
169
170     String endInput = tokens.dequeue();
171     Reporter.assertElseFatalError(endInput.equals("### END OF INPUT ###"),
172         "Recieved: " + endInput + " , Expected: ### END OF INPUT ###");
173
174     // update program name, context, and body if all syntax passes
175     this.setName(identifier);
176     this.swapContext(context);
177     this.swapBody(body);
178
179 }
180
181 /*
182 * Main test method -----
183 */
184
185 /**
186 * Main method.
187 */

```

```
188     * @param args
189     *         the command line arguments
190     */
191     public static void main(String[] args) {
192         SimpleReader in = new SimpleReader1L();
193         SimpleWriter out = new SimpleWriter1L();
194         /*
195          * Get input file name
196          */
197         out.print("Enter valid BL program file name: ");
198         String fileName = in.nextLine();
199         /*
200          * Parse input file
201          */
202         out.println("*** Parsing input file ***");
203         Program p = new Program1Parse1();
204         SimpleReader file = new SimpleReader1L(fileName);
205         Queue<String> tokens = Tokenizer.tokens(file);
206         file.close();
207         p.parse(tokens);
208         /*
209          * Pretty print the program
210          */
211         out.println("*** Pretty print of parsed program ***");
212         p.prettyPrint(out);
213
214         in.close();
215         out.close();
216     }
217
218 }
219
```