



THE OHIO STATE  
UNIVERSITY



# PROJECT 8: PROGRAM AND STATEMENT PARSE

Daniil Gofman

Ansh Pachauri

SW 2: Dev & Dsgn

Paolo Bucci

Yiyang Chen

Shivam Gupta

November 14, 2023

```
1 import components.map.Map;
2 import components.map.Map.Pair;
3 import components.program.Program;
4 import components.program.Program1;
5 import components.queue.Queue;
6 import components.simplereader.SimpleReader;
7 import components.simplereader.SimpleReader1L;
8 import components.simplewriter.SimpleWriter;
9 import components.simplewriter.SimpleWriter1L;
10 import components.statement.Statement;
11 import components.utilities.Reporter;
12 import components.utilities.Tokenizer;
13
14 /**
15  * Layered implementation of secondary method {@code parse} for {@code
16  * Program}.
17  * @author Daniil Gofman and Ansh Pachauri
18  *
19  */
20 public final class Program1Parse1 extends Program1 {
21
22     /*
23      * Private members
24      */
25
26     /**
27      * Parses a single BL instruction from {@code tokens} returning the
28      * instruction name as the value of the function and the body of the
29      * instruction in {@code body}.
30      *
31      * @param tokens
32      *         the input tokens
33      * @param body
34      *         the instruction body
35      * @return the instruction name
36      * @replaces body
37      * @updates tokens
38      * @requires <pre>
39      *   [<"INSTRUCTION"> is a prefix of tokens] and
40      *   [<Tokenizer.END_OF_INPUT> is a suffix of tokens]
41      * </pre>
42      * @ensures <pre>
43      *   if [an instruction string is a proper prefix of #tokens] and
44      *   [the beginning name of this instruction equals its ending name]
45      *   and
46      *   [the name of this instruction does not equal the name of a
47      *   primitive
48      *   instruction in the BL language] then
```

```

47      *   parseInstruction = [name of instruction at start of #tokens] and
48      *   body = [Statement corresponding to the block string that is the body
of
49      *           the instruction string at start of #tokens] and
50      *   #tokens = [instruction string at start of #tokens] * tokens
51      * else
52      *   [report an appropriate error message to the console and terminate
client]
53      * </pre>
54      */
55      private static String parseInstruction(Queue<String> tokens,
56          Statement body) {
57          assert tokens != null : "Violation of: tokens is not null";
58          assert body != null : "Violation of: body is not null";
59          assert tokens.length() > 0 && tokens.front().equals("INSTRUCTION") :
""
60          + "Violation of: <\\"INSTRUCTION\\"> is proper prefix of
tokens";
61          //check the keyword INSTRUCTION.
62          String inst = tokens.dequeue();
63          Reporter.assertElseFatalError(inst.equals("INSTRUCTION"),
64              "Error: \\"INSTRUCTION\\" not found");
65          //check the instruction's name. If it's equal to primitive
66          //instructions, send error message.
67          String instName = tokens.dequeue();
68          boolean name = !instName.equals("move") && !instName.equals
("turnleft")
69              && !instName.equals("turnright") && !instName.equals
("infect")
70              && !instName.equals("skip");
71          Reporter.assertElseFatalError(name,
72              "Error: intruction name cannot be a primitive instruction");
73          //check the keyword IS.
74          String is = tokens.dequeue();
75          Reporter.assertElseFatalError(is.equals("IS"),
76              "Error: \\"IS\\" not found");
77          //parse the block after the keywords.
78          body.parseBlock(tokens);
79          //check the keyword END.
80          String end = tokens.dequeue();
81          Reporter.assertElseFatalError(end.equals("END"),
82              "Error: \\"END\\" not found");
83          //check the instruction's name. If it's not the same as the
instruction
84          //name at the last instruction, send error message.
85          String endInstName = tokens.dequeue();
86          Reporter.assertElseFatalError(endInstName.equals(instName), "Error:
\\"
87              + endInstName + "\\" is not equal to \\"" + instName + "\\"");
88

```

```
89         return instName;
90     }
91     /*
92     * Constructors
93     */
94
95     /**
96     * No-argument constructor.
97     */
98     public Program1Parse1() {
99         super();
100     }
101
102     /*
103     * Public methods
104     */
105
106     @Override
107     public void parse(SimpleReader in) {
108         assert in != null : "Violation of: in is not null";
109         assert in.isOpen() : "Violation of: in.is_open";
110         Queue<String> tokens = Tokenizer.tokens(in);
111         this.parse(tokens);
112     }
113
114     @Override
115     public void parse(Queue<String> tokens) {
116         assert tokens != null : "Violation of: tokens is not null";
117         assert tokens.length() > 0 : ""
118             + "Violation of: Tokenizer.END_OF_INPUT is a suffix of
119 tokens";
120         //check the keyword PROGRAM.
121         String program = tokens.dequeue();
122         Reporter.assertElseFatalError(program.equals("PROGRAM"),
123             "Error: \"PROGRAM\" not found");
124         //check the program's name. If it's equal to primitive
125         //instructions, send error message.
126         String programName = tokens.dequeue();
127         boolean name = !programName.equals("move")
128             && !programName.equals("turnleft")
129             && !programName.equals("turnright")
130             && !programName.equals("infect") && !programName.equals
131 ("skip");
132         Reporter.assertElseFatalError(name,
133             "Error: intruction name cannot be a primitive instruction");
134         //check the keyword IS.
135         String is = tokens.dequeue();
136         Reporter.assertElseFatalError(is.equals("IS"),
```

```
135         "Error: \"IS\" not found");
136     //create a map to be the context of the program.
137     Map<String, Statement> programCnxt = this.newContext();
138     //check whether the next part is an instruction or the body.
139     String firstToken = tokens.front();
140     while (firstToken.equals("INSTRUCTION")) {
141         Statement instBody = this.newBody();
142         String instName = parseInstruction(tokens, instBody);
143         //create a body for the current instruction, and parse the
144         //instruction. Check if the current instruction was already
        defined before.
145         for (Pair<String, Statement> context : programCnxt) {
146             Reporter.assertElseFatalError(!context.key().equals
(instName),
147                 "Error: the instruction \"" + instName
148                     + "\" is already defined");
149         }
150         //add the instruction to the context.
151         programCnxt.add(instName, instBody);
152         //change the string to next line.
153         firstToken = tokens.front();
154     }
155     //check the keyword BEGIN.
156     String begin = tokens.dequeue();
157     Reporter.assertElseFatalError(begin.equals("BEGIN"),
158         "Error: \"BEGIN\" not found");
159     //create a new statement to be the body of the program.
160     Statement programBody = this.newBody();
161     //parse the block after the keywords.
162     programBody.parseBlock(tokens);
163     //check the keyword END.
164     String end = tokens.dequeue();
165     Reporter.assertElseFatalError(end.equals("END"),
166         "Error: \"END\" not found");
167     //check the program's name. If it's not the same as the program name
168     //at the beginning of the BL program, send error message.
169     String endProgramName = tokens.dequeue();
170     Reporter.assertElseFatalError(endProgramName.equals(programName),
171         "Error: \"" + endProgramName + "\" is not equal to \""
172             + programName + "\"");
173     //check whether the last token is ### END OF INPUT ### or not.
174     String endOfInput = tokens.dequeue();
175     Reporter.assertElseFatalError(endOfInput.equals("### END OF INPUT
###"),
176         "Error: \"### END OF INPUT ###\" not found");
177
178     this.setName(programName);
179     this.swapContext(programCnxt);
180     this.swapBody(programBody);
181
```

```
182     }
183
184     /*
185     * Main test method
186     */
187
188     /**
189     * Main method.
190     *
191     * @param args
192     *         the command line arguments
193     */
194     public static void main(String[] args) {
195         SimpleReader in = new SimpleReader1L();
196         SimpleWriter out = new SimpleWriter1L();
197         /*
198         * Get input file name
199         */
200         out.print("Enter valid BL program file name: ");
201         String fileName = in.nextLine();
202         /*
203         * Parse input file
204         */
205         out.println("*** Parsing input file ***");
206         Program p = new Program1Parse1();
207         SimpleReader file = new SimpleReader1L(fileName);
208         Queue<String> tokens = Tokenizer.tokens(file);
209         file.close();
210         p.parse(tokens);
211         /*
212         * Pretty print the program
213         */
214         out.println("*** Pretty print of parsed program ***");
215         p.prettyPrint(out);
216
217         in.close();
218         out.close();
219     }
220
221 }
222
```