```java
import components.map.Map;

/**
 * Layered implementation of secondary method {@code parse} for {@code Program}.
 *
 * @author Qinuo Shi & Yiming Cheng
 *
 */
public final class Program1Parse1 extends Program1 {

    /*
     * Private members --------------------------------------------------------
     */

    /**
     * Parses a single BL instruction from {@code tokens} returning the
     * instruction name as the value of the function and the body of the
     * instruction in {@code body}.
     *
     * @param tokens
     *            the input tokens
     * @param body
     *            the instruction body
     * @return the instruction name
     * @replaces body
     * @updates tokens
     * @requires <pre>
     * [<"INSTRUCTION"> is a prefix of tokens]  and
     *  [<Tokenizer.END_OF_INPUT> is a suffix of tokens]
     * </pre>
     * @ensures <pre>
     * if [an instruction string is a proper prefix of #tokens]  and
     *    [the beginning name of this instruction equals its ending name]  and
     *    [the name of this instruction does not equal the name of a primitive
     *     instruction in the BL language] then
     *  parseInstruction = [name of instruction at start of #tokens]  and
     *  body = [Statement corresponding to the block string that is the body of
     *          the instruction string at start of #tokens]  and
     *  #tokens = [instruction string at start of #tokens] * tokens
     * else
     *  [report an appropriate error message to the console and terminate client]
     * </pre>
     */
    private static String parseInstruction(Queue<String> tokens,
            Statement body) {
        assert tokens != null : "Violation of: tokens is not null";
        assert body != null : "Violation of: body is not null";
        assert tokens.length() > 0 && tokens.front().equals("INSTRUCTION") : ""
                + "Violation of: <\"INSTRUCTION\"> is proper prefix of tokens";

        // TODO - fill in body
        tokens.dequeue();

        /*
         * Parse INSTRUCTION
         */
        /*
```

```
         * Check for errors according to the order of BL languages.
         */
        /*
         * If the BL format is not found, report an error here
         */
        String instrName = tokens.dequeue();
        Reporter.assertElseFatalError(Tokenizer.isIdentifier(instrName),
                "Name is identifier");

        Reporter.assertElseFatalError(
                !instrName.equals("move") && !instrName.equals("turnleft")
                        && !instrName.equals("turnright")
                        && !instrName.equals("infect")
                        && !instrName.equals("skip"),
                "Name not primitive instruction");

        Reporter.assertElseFatalError(tokens.dequeue().equals("IS"),
                "Cannot find IS");

        body.parseBlock(tokens);
        Reporter.assertElseFatalError(tokens.dequeue().equals("END"),
                "Cannot find END");

        String check = tokens.dequeue();
        Reporter.assertElseFatalError(instrName.equals(check),
                "Beginning of instruction equals its ending name");

        return instrName;
    }

    /*
     * Constructors -----------------------------------------------------------
     */

    /**
     * No-argument constructor.
     */
    public Program1Parse1() {
        super();
    }

    /*
     * Public methods ---------------------------------------------------------
     */

    @Override
    public void parse(SimpleReader in) {
        assert in != null : "Violation of: in is not null";
        assert in.isOpen() : "Violation of: in.is_open";
        Queue<String> tokens = Tokenizer.tokens(in);
        this.parse(tokens);
    }

    @Override
    public void parse(Queue<String> tokens) {
        assert tokens != null : "Violation of: tokens is not null";
        assert tokens.length() > 0 : ""
```

```
                + "Violation of: Tokenizer.END_OF_INPUT is a suffix of tokens";

        // TODO - fill in body
        /*
         * If the BL format is not found, report an error here
         */
        Map<String, Statement> map = this.newContext();
        Statement toolBody = this.newBody();
        Reporter.assertElseFatalError(tokens.dequeue().equals("PROGRAM"),
                "Cannot find PROGRAM at the beginning.");

        /*
         * Extract the name
         */
        String name = tokens.dequeue();
        Reporter.assertElseFatalError(Tokenizer.isIdentifier(name),
                "Name is identifier");

        /*
         * If the BL format is not found, report an error here
         */
        this.setName(name);
        Reporter.assertElseFatalError(tokens.dequeue().equals("IS"),
                "Cannot find IS");

        /*
         * Parse the INSTRUCTION
         */
        while (tokens.front().equals("INSTRUCTION")) {
            Statement toolContext = this.newBody();
            String instrName = parseInstruction(tokens, toolContext);
            Reporter.assertElseFatalError(!map.hasKey(instrName),
                    "It is already existed.");
            map.add(instrName, toolContext);
        }

        /*
         * If the BL format is not found, report an error here
         */
        this.swapContext(map);
        Reporter.assertElseFatalError(tokens.dequeue().equals("BEGIN"),
                "Cannot find BEGIN");

        /*
         * If the BL format is not found, report an error here
         */
        toolBody.parseBlock(tokens);
        this.swapBody(toolBody);
        Reporter.assertElseFatalError(tokens.dequeue().equals("END"),
                "Cannot find END");

        String nameInQueue = tokens.dequeue();
        Reporter.assertElseFatalError(name.equals(nameInQueue),
                "Beginning identifier equals ending identifier");

        Reporter.assertElseFatalError(
                tokens.front().equals(Tokenizer.END_OF_INPUT),
                "Tokenizer.END_OF_INPUT is a suffix of tokens");

    }

    /*
     * Main test method -------------------------------------------------------
     */

    /**
     * Main method.
     *
     * @param args
     *            the command line arguments
     */
    public static void main(String[] args) {
        SimpleReader in = new SimpleReader1L();
        SimpleWriter out = new SimpleWriter1L();
        /*
         * Get input file name
         */
        out.print("Enter valid BL program file name: ");
        String fileName = in.nextLine();
        /*
         * Parse input file
         */
        out.println("*** Parsing input file ***");
        Program p = new Program1Parse1();
        SimpleReader file = new SimpleReader1L(fileName);
        Queue<String> tokens = Tokenizer.tokens(file);
        file.close();
        p.parse(tokens);
        /*
         * Pretty print the program
         */
        out.println("*** Pretty print of parsed program ***");
        p.prettyPrint(out);

        in.close();
        out.close();
    }

}

```