```java
import components.map.Map;

/**
 * Layered implementation of secondary method {@code parse} for {@code Program}.
 *
 * @author Gabe Azzarita and Ty Fredrick
 *
 */
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";

        // check sytanx for INSTRUCTION
        String instruction = tokens.dequeue();
        Reporter.assertElseFatalError(instruction.equals("INSTRUCTION"),
                "Recieved: " + instruction + ", Expected: INSTRUCTION.");

        // check for proper identifier
        String identifier = tokens.dequeue();
        Reporter.assertElseFatalError(Tokenizer.isIdentifier(identifier),
                identifier + ": invalid identifier.");

        // check syntax for IS
        String is = tokens.dequeue();
        Reporter.assertElseFatalError(is.equals("IS"),
                "Recieved: " + is + ", Expected: IS.");

        // parse body
        body.parseBlock(tokens);

        // check syntax for END and identifier
        String end = tokens.dequeue();
        Reporter.assertElseFatalError(end.equals("END"),
                "Recieved: " + end + " , Expected: END.");

        Reporter.assertElseFatalError(tokens.dequeue().equals(identifier),
                "Identifier at start does not match identifier at end.");

        return identifier;
    }

    /*
     * 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";

        // check syntax for PROGRAM, identifier, and IS
        String program = tokens.dequeue();
        Reporter.assertElseFatalError(program.equals("PROGRAM"),
                "Recieved: " + program + ", expected: PROGRAM.");

        String identifier = tokens.dequeue();
        Reporter.assertElseFatalError(Tokenizer.isIdentifier(identifier),
                identifier + ": invalid identifier.");

        String is = tokens.dequeue();
        Reporter.assertElseFatalError(is.equals("IS"),
                "Recieved: " + is + ", Expected: IS.");

        /*
         * create and fill context so long as front token is INSTRUCTION and the
         * instruction name is not already defined in the map
         */

        Map<String, Statement> context = this.newContext();
        if (tokens.length() > 0) {

            while (tokens.front().equals("INSTRUCTION")) {

                Statement block = this.newBody();
                String instructionName = parseInstruction(tokens, block);

                Reporter.assertElseFatalError(!context.hasKey(instructionName),
                        "Cannot have duplicate user defined instructions.");

                context.add(instructionName, block);

            }
        }

        // check syntax
        String begin = tokens.dequeue();
        Reporter.assertElseFatalError(begin.equals("BEGIN"),
                "Recieved: " + begin + ", Expected: BEGIN");

        // create and parse new body
        Statement body = this.newBody();
        body.parseBlock(tokens);

        // check syntax for END and identifier and END OF INPUT
        String end = tokens.dequeue();
        Reporter.assertElseFatalError(end.equals("END"),
                "Recieved: " + end + " , Expected: END.");

        Reporter.assertElseFatalError(tokens.dequeue().equals(identifier),
                "Identifier at start does not match identifier at end.");

        String endInput = tokens.dequeue();
        Reporter.assertElseFatalError(endInput.equals("### END OF INPUT ###"),
                "Recieved: " + endInput + " , Expected: ### END OF INPUT ###");

        // update program name, context, and body if all syntax passes
        this.setName(identifier);
        this.swapContext(context);
        this.swapBody(body);

    }

    /*
     * 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();
    }

}

```