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

```java
import components.map.Map;
import components.map.Map.Pair;
import components.program.Program;
import components.program.Program1;
import components.queue.Queue;
import components.simplereader.SimpleReader;
import components.simplereader.SimpleReader1L;
import components.simplewriter.SimpleWriter;
import components.simplewriter.SimpleWriter1L;
import components.statement.Statement;
import components.utilities.Reporter;
import components.utilities.Tokenizer;

/**
 * Layered implementation of secondary method {@code parse} for {@code
Program}.
 *
 * @author Daniil Gofman and Ansh Pachauri
 *
 */
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 the keyword INSTRUCTION.
        String inst = tokens.dequeue();
        Reporter.assertElseFatalError(inst.equals("INSTRUCTION"),
                "Error: \"INSTRUCTION\" not found");
        //check the instruction's name. If it's equal to primitive
        //instructions, send error message.
        String instName = tokens.dequeue();
        boolean name = !instName.equals("move") && !instName.equals("turnleft")
                && !instName.equals("turnright") && !instName.equals("infect")
                && !instName.equals("skip");
        Reporter.assertElseFatalError(name,
                "Error: intruction name cannot be a primitive instruction");
        //check the keyword IS.
        String is = tokens.dequeue();
        Reporter.assertElseFatalError(is.equals("IS"),
                "Error: \"IS\" not found");
        //parse the block after the keywords.
        body.parseBlock(tokens);
        //check the keyword END.
        String end = tokens.dequeue();
        Reporter.assertElseFatalError(end.equals("END"),
                "Error: \"END\" not found");
        //check the instruction's name. If it's not the same as the instruction
        //name at the last instruction, send error message.
        String endInstName = tokens.dequeue();
        Reporter.assertElseFatalError(endInstName.equals(instName), "Error: \""
                + endInstName + "\" is not equal to \"" + instName + "\"");

        return instName;
    }
    /*
     * 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 the keyword PROGRAM.
        String program = tokens.dequeue();
        Reporter.assertElseFatalError(program.equals("PROGRAM"),
                "Error: \"PROGRAM\" not found");
        //check the program's name. If it's equal to primitive
        //instructions, send error message.
        String programName = tokens.dequeue();
        boolean name = !programName.equals("move")
                && !programName.equals("turnleft")
                && !programName.equals("turnright")
                && !programName.equals("infect") && !programName.equals("skip");
        Reporter.assertElseFatalError(name,
                "Error: intruction name cannot be a primitive instruction");
        //check the keyword IS.
        String is = tokens.dequeue();
        Reporter.assertElseFatalError(is.equals("IS"),
                "Error: \"IS\" not found");
        //create a map to be the context of the program.
        Map<String, Statement> programCnxt = this.newContext();
        //check whether the next part is an instruction or the body.
        String firstToken = tokens.front();
        while (firstToken.equals("INSTRUCTION")) {
            Statement instBody = this.newBody();
            String instName = parseInstruction(tokens, instBody);
            //create a body for the current instruction, and parse the
            //instruction. Check if the current instruction was already
   defined before.
            for (Pair<String, Statement> context : programCnxt) {
                Reporter.assertElseFatalError(!context.key().equals
   (instName),
                        "Error: the instruction \"" + instName
                                + "\" is already defined");
            }
            //add the instruction to the context.
            programCnxt.add(instName, instBody);
            //change the string to next line.
            firstToken = tokens.front();
        }
        //check the keyword BEGIN.
        String begin = tokens.dequeue();
        Reporter.assertElseFatalError(begin.equals("BEGIN"),
                "Error: \"BEGIN\" not found");
        //create a new statement to be the body of the program.
        Statement programBody = this.newBody();
        //parse the block after the keywords.
        programBody.parseBlock(tokens);
        //check the keyword END.
        String end = tokens.dequeue();
        Reporter.assertElseFatalError(end.equals("END"),
                "Error: \"END\" not found");
        //check the program's name. If it's not the same as the program name
        //at the beginning of the BL program, send error message.
        String endProgramName = tokens.dequeue();
        Reporter.assertElseFatalError(endProgramName.equals(programName),
                "Error: \"" + endProgramName + "\" is not equal to \""
                        + programName + "\"");
        //check whether the last token is ### END OF INPUT ### or not.
        String endOfInput = tokens.dequeue();
        Reporter.assertElseFatalError(endOfInput.equals("### END OF INPUT
   ###"),
                "Error: \"### END OF INPUT ###\" not found");

        this.setName(programName);
        this.swapContext(programCnxt);
        this.swapBody(programBody);

```

```
    }

    /*
     * 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();
    }

}

```