CSE 2231 – Software 2: Software Development and Design

Professor: Rob LaTour

Project #8: Program and Statement Parser Implementation(s)

Date of Submission: April 7th, 2023

The Ohio State University

College of Engineering

Columbus, Ohio

```
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 Danny Kan (kan.74@osu.edu)

 * @author Jatin Mamtani (mamtani.6@osu.edu)

 *

 */

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";
        String instructionToken = tokens.dequeue();
        Reporter.assertElseFatalError(instructionToken.equals("INSTRUCTION"),
                "Error:\n------\nExpected: " + "\"" + "INSTRUCTION" + "\"");
            String[] primitiveInstructions = { "move", "turnleft", "turnright",
                    "infect", "skip" };
            String instructionIdentifier1 = tokens.dequeue();
            Reporter.assertElseFatalError(
                    !Tokenizer.isKeyword(instructionIdentifier1),
                    "Error:\n------\n");
            Reporter.assertElseFatalError(
                    Tokenizer.isIdentifier(instructionIdentifier1),
                    "Error:\n------\n");
            for (String x : primitiveInstructions) {
                Reporter.assertElseFatalError(!x.equals(instructionIdentifier1),
                        "Error:\n------\nThe name of each new user-defined instruction must not be the name of one of the
    primitive instructions, i.e., move, turnleft, turnright, infect, or skip.");
            }
            Reporter.assertElseFatalError(tokens.dequeue().equals("IS"),
                    "Error:\n------\nExpected: " + "\"" + "IS" + "\"");
            body.parseBlock(tokens);
            Reporter.assertElseFatalError(tokens.dequeue().equals("END"),
                    "Error:\n------\nExpected: " + "\"" + "END" + "\"");
            String instructionIdentifier2 = tokens.dequeue();
            Reporter.assertElseFatalError(
                    instructionIdentifier2.equals(instructionIdentifier1),
                    "Error:\n------\nThe identifier at the end of each new instruction definition must be the same as the
    identifier at the beginning of the definition.");
            return instructionIdentifier1;
        }


        /*
         * 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";
    Program program = new Program1Parse1();
    String programToken = tokens.dequeue();
    Reporter.assertElseFatalError(programToken.equals("PROGRAM"),
            "Error:\n------\nExpected: " + "\"" + "PROGRAM" + "\"");
    String programIdentifier1 = tokens.dequeue();
    // need to make sure the program name is not a keyword.
    Reporter.assertElseFatalError(!Tokenizer.isKeyword(programIdentifier1),
            "Error:\n------\nThe program name is a keyword.");
    // need to make sure the program name is a valid identifier.
```

```
        Reporter.assertElseFatalError(
                Tokenizer.isIdentifier(programIdentifier1),
                "Error:\n------\nThe program name is not a valid identifier.");
        Reporter.assertElseFatalError(tokens.dequeue().equals("IS"),
                "Error:\n------\nExpected: " + "\"" + "IS" + "\"");
        Map<String, Statement> context = program.newContext();
        String instr = tokens.front();
        while (instr.equals("INSTRUCTION")) {
            Statement body = program.newBody();
            String name = parseInstruction(tokens, body);
            for (Pair<String, Statement> x : context) {
                Reporter.assertElseFatalError(!x.key().equals(name),
                        "Error:\n------\nThe name of each new user-defined instruction must be unique, i.e., there cannot be two user-defined instructions with the same name.");
            }
            context.add(name, body);
            instr = tokens.front();
        }
        Reporter.assertElseFatalError(instr.equals("BEGIN"),
                "Error:\n------\nExpected: " + "\"" + "BEGIN" + "\"");
        instr = tokens.dequeue(); // ...
        Statement pBody = program.newBody();
        pBody.parseBlock(tokens);
        Reporter.assertElseFatalError(tokens.dequeue().equals("END"),
                "Error:\n------\nExpected: " + "\"" + "END" + "\"");
        String programIdentifier2 = tokens.dequeue();
        Reporter.assertElseFatalError(
                programIdentifier2.equals(programIdentifier1),
                "Error:\n------\nThe identifier at the end of the program must be the same as the identifier at the beginning of the program.");
        Reporter.assertElseFatalError(
                tokens.front().equals("### END OF INPUT ###"),
            "Error:\n------\nExpected: ### END OF INPUT ###");
    this.setName(programIdentifier1);
    this.swapBody(pBody);
    this.swapContext(context);
}

/*
 * 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();
    }

}
```