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.queue.Queue;
import components.simplereader.SimpleReader;
import components.simplereader.SimpleReader1L;
import components.simplewriter.SimpleWriter;
import components.simplewriter.SimpleWriter1L;
import components.statement.Statement;
import components.statement.Statement1;
import components.utilities.Reporter;
import components.utilities.Tokenizer;

/**
 * Layered implementation of secondary methods {@code parse} and
 * {@code parseBlock} for {@code Statement}.
 *
 * @author Danny Kan (kan.74@osu.edu)
 * @author Jatin Mamtani (mamtani.6@osu.edu)
 *
 */
public final class Statement1Parse1 extends Statement1 {

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

    /**
     * Converts {@code c} into the corresponding {@code Condition}.
     *
     * @param c
     *            the condition to convert
     * @return the {@code Condition} corresponding to {@code c}
     * @requires [c is a condition string]
     * @ensures parseCondition = [Condition corresponding to c]
     */
    private static Condition parseCondition(String c) {
        assert c != null : "Violation of: c is not null";
        assert Tokenizer
                .isCondition(c) : "Violation of: c is a condition string";
        return Condition.valueOf(c.replace('-', '_').toUpperCase());
    }

    /**
     * Parses an IF or IF_ELSE statement from {@code tokens} into {@code s}.
     *
     * @param tokens
     *            the input tokens
     * @param s
     *            the parsed statement
     * @replaces s
     * @updates tokens
     * @requires <pre>
     * [<"IF"> is a prefix of tokens]  and
     *  [<Tokenizer.END_OF_INPUT> is a suffix of tokens]
     * </pre>
     * @ensures <pre>
     * if [an if string is a proper prefix of #tokens] then
     *  s = [IF or IF_ELSE Statement corresponding to if string at start of #tokens]  and
     *  #tokens = [if string at start of #tokens] * tokens
     * else
     *  [reports an appropriate error message to the console and terminates client]
     * </pre>
     */
    private static void parseIf(Queue<String> tokens, Statement s) {
        assert tokens != null : "Violation of: tokens is not null";
        assert s != null : "Violation of: s is not null";
    assert tokens.length() > 0 && tokens.front().equals("IF") : ""
            + "Violation of: <\"IF\"> is proper prefix of tokens";
    String ifToken = tokens.dequeue();
    Reporter.assertElseFatalError(Tokenizer.isCondition(tokens.front()),
            "Error\n------\nNot a valid condition.");
    String conditionToken = tokens.dequeue();
    Condition ifCondition = parseCondition(conditionToken);
    Reporter.assertElseFatalError(tokens.front().equals("THEN"),
            "Error:\n------\nExpected: " + "\"" + "THEN" + "\"");
    String thenToken = tokens.dequeue();
    Statement ifBlock = s.newInstance();
    ifBlock.parseBlock(tokens);
    // error message.
    Reporter.assertElseFatalError(
            tokens.front().equals("ELSE") || tokens.front().equals("END"),
            "...");
    if (tokens.front().equals("ELSE")) {
        String elseToken = tokens.dequeue();
        Statement elseBlock = s.newInstance();
        elseBlock.parseBlock(tokens);
        s.assembleIfElse(ifCondition, ifBlock, elseBlock);
        Reporter.assertElseFatalError(tokens.front().equals("END"),
                "Error:\n------\nExpected: " + "\"" + "END" + "\"");
        String endToken = tokens.dequeue();
    } else {
        s.assembleIf(ifCondition, ifBlock);
        Reporter.assertElseFatalError(tokens.front().equals("END"),
                "Error:\n------\nExpected: " + "\"" + "END" + "\"");
        String end = tokens.dequeue();
    }
    String endIfToken = tokens.dequeue();
    // error message.
```

```
    Reporter.assertElseFatalError(endIfToken.equals("IF"), "...");
}

/**
 * Parses a WHILE statement from {@code tokens} into {@code s}.
 *
 * @param tokens
 *            the input tokens
 * @param s
 *            the parsed statement
 * @replaces s
 * @updates tokens
 * @requires <pre>
 * [<"WHILE"> is a prefix of tokens]  and
 *  [<Tokenizer.END_OF_INPUT> is a suffix of tokens]
 * </pre>
 * @ensures <pre>
 * if [a while string is a proper prefix of #tokens] then
 *  s = [WHILE Statement corresponding to while string at start of #tokens]  and
 *  #tokens = [while string at start of #tokens] * tokens
 * else
 *  [reports an appropriate error message to the console and terminates client]
 * </pre>
 */
private static void parseWhile(Queue<String> tokens, Statement s) {
    assert tokens != null : "Violation of: tokens is not null";
    assert s != null : "Violation of: s is not null";
    assert tokens.length() > 0 && tokens.front().equals("WHILE") : ""
            + "Violation of: <\"WHILE\"> is proper prefix of tokens";
    String whileToken = tokens.dequeue();
    Reporter.assertElseFatalError(Tokenizer.isCondition(tokens.front()),
            "Error:\n------\nNot a valid condition.");
    String conditionToken = tokens.dequeue();
    Condition whileCondition = parseCondition(conditionToken);
    Reporter.assertElseFatalError(tokens.front().equals("DO"),
            "Error:\n------\nExpected: " + "\"" + "DO" + "\"");
    String doToken = tokens.dequeue();
    Statement whileBlock = s.newInstance();
    whileBlock.parseBlock(tokens);
    // error message.
    Reporter.assertElseFatalError(tokens.dequeue().equals("END")
            && tokens.dequeue().equals("WHILE"), "...");
    s.assembleWhile(whileCondition, whileBlock);
}

/**
 * Parses a CALL statement from {@code tokens} into {@code s}.
 *
 * @param tokens
 *            the input tokens
 * @param s
 *            the parsed statement
 * @replaces s
 * @updates tokens
 * @requires [identifier string is a proper prefix of tokens]
 * @ensures <pre>
 * s =
 *   [CALL Statement corresponding to identifier string at start of #tokens]  and
 *  #tokens = [identifier string at start of #tokens] * tokens
 * </pre>
 */
private static void parseCall(Queue<String> tokens, Statement s) {
    assert tokens != null : "Violation of: tokens is not null";
    assert s != null : "Violation of: s is not null";
    assert tokens.length() > 0
            && Tokenizer.isIdentifier(tokens.front()) : ""
                    + "Violation of: identifier string is proper prefix of tokens";
    String identifier = tokens.dequeue();
    s.assembleCall(identifier);
}

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

/**
 * No-argument constructor.
 */
public Statement1Parse1() {
    super();
}

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

@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";
    if (tokens.front().equals("IF")) {
        parseIf(tokens, this);
    } else if (tokens.front().equals("WHILE")) {
        parseWhile(tokens, this);
    } else if (Tokenizer.isIdentifier(tokens.front())) {
        parseCall(tokens, this);
    } else {
        // error message.
        Reporter.assertElseFatalError(tokens.front().equals("IF")
                || tokens.front().equals("WHILE")
                || Tokenizer.isIdentifier(tokens.front()), "...");
    }
}

@Override
public void parseBlock(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";
    this.clear();
    String token = tokens.front();
    int i = 0;
    while (Tokenizer.isIdentifier(token) || token.equals("IF")
            || token.equals("WHILE")) {
        Statement st = this.newInstance();
        st.parse(tokens);
        this.addToBlock(i, st);
        token = tokens.front();
        i++;
    }
}

/*
 * 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 statement(s) file name: ");
    String fileName = in.nextLine();
    /*
     * Parse input file
     */
    out.println("*** Parsing input file ***");
    Statement s = new Statement1Parse1();
    SimpleReader file = new SimpleReader1L(fileName);
    Queue<String> tokens = Tokenizer.tokens(file);
    file.close();
    s.parse(tokens); // replace with parseBlock to test other method
    /*
     * Pretty print the statement(s)
     */
    out.println("*** Pretty print of parsed statement(s) ***");
    s.prettyPrint(out, 0);

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

```
}
```