Raul Aguilar

Prof. Paulding

CS 220

November 1, 2019

Homework 8: Virtual Machine I

VirtualMachine.java

/\*

    @author Raul Aguilar

    @date   October 26, 2019

\*/

import java.util.Scanner;

public class VirtualMachine {

    public static void main(String[] args) {

        Scanner keyboard = new Scanner(System.in);

        Parser parser = new Parser();

        CodeWriter codeWriter = new CodeWriter();

        String inputFileName, outputFileName;

        // Open file from command line or console

        if(args.length == 1) {

            System.out.println("command line arg = " + args[0]);

            inputFileName = args[0];

        } else {

            System.out.println("Please enter assembly file name you would like to assemble.");

            System.out.println("Don't forget the .vm extension: ");

            inputFileName = keyboard.nextLine();

            keyboard.close();

        }

        outputFileName = inputFileName.substring(0, inputFileName.lastIndexOf('.')) + ".asm";

        codeWriter.CodeWriter(outputFileName);

        // Driver

        parser.Parser(inputFileName);

        while(parser.hasMoreCommands()) {

            parser.advance();

            // write to file

            if(parser.getCommandType() == Command.C\_ARITHMETIC) {

                codeWriter.writeArithmetic(parser.getArg1());

            } else if (parser.getCommandType() == Command.C\_PUSH || parser.getCommandType() == Command.C\_POP) {

                codeWriter.writePushPop(parser.getCommandType(), parser.getArg1(), parser.getArg2());

            }

        }

        codeWriter.writeInfiniteLoop();

        codeWriter.close();

        System.out.println("Finished assemblng. Program exiting.");

    }

}

Parser.java

/\*\*

 \* @author  Raul Aguilar

 \* @date    October 23, 2019

 \* Parser: Handles the parsing of a single .vm file, and encapsulates access to the input code.

 \*  It reads VM commands, parses them, and provides convenient access to their components. In

 \*  addition, it removes all white spaces and comments.

 \*/

import java.io.FileNotFoundException;

import java.io.FileReader;

import java.util.Scanner;

public class Parser {

    private Scanner inputFile;

    private String[] commands;

    private int lineNumber;

    private String rawCommand, cleanCommand;

    private Command commandType;

    private String arg1 = "";

    private int arg2;

    /\*\*

     \* Opens the input file and gets ready to parse it

     \* @param fileName Name of the vm file

     \*/

    public void Parser(String fileName) {

        try {

            inputFile = new Scanner(new FileReader(fileName));

        } catch (FileNotFoundException e) {

            System.err.println("File could not be found. Exiting program.");

            System.exit(0);

        }

    }

    /\*\*

     \* Returns boolean if there are more commands in the file, if not closes

     \* the file

     \* @return True if there are more commands, otherwise false and closes stream

     \*/

    public boolean hasMoreCommands() {

        if(inputFile.hasNextLine()) {

            return true;

        } else {

            inputFile.close();

            return false;

        }

    }

    /\*\*

     \* Reads the next command from the input and makes it the current command. Should

     \* be called only if hasMoreCommands() is true. Initially there is no current

     \* command.

     \*/

    public void advance() {

        lineNumber++;

        rawCommand = inputFile.nextLine();

        cleanLine();

        parseCommand();

        parseCommandType();

        if(commandType != Command.NO\_COMMAND) {

            parseArg1();

            if(commandType == Command.C\_PUSH || commandType == Command.C\_POP) {

                parseArg2();

            }

        }

    }

    /\*\*

     \* Reads command line from vm file and strips it of whitespaces and comments

     \*/

    private void cleanLine() {

        int commentIndex;

        if(rawCommand == null) {

            cleanCommand = "";

        } else {

            commentIndex = rawCommand.indexOf("/");

            if(commentIndex != -1) {

                cleanCommand = rawCommand.substring(0, commentIndex);

                cleanCommand = cleanCommand.replaceAll(" ", "");

                cleanCommand = cleanCommand.replaceAll("\t", "");

            } else {

                cleanCommand = rawCommand;

            }

        }

    }

    /\*\*

     \* Parse the cleaned up line into parts using a String array

     \*/

    private void parseCommand() {

        if(cleanCommand != null) {

            commands = cleanCommand.split(" ");

        }

    }

    /\*\*

     \* Guess the command type

     \*/

    private void parseCommandType() {

        if(cleanCommand == null || cleanCommand.length() == 0) {

            commandType = Command.NO\_COMMAND;

        } else if(commands.length == 1) {

            commandType = Command.C\_ARITHMETIC;

        } else if(commands[0].equals("pop")) {

            commandType = Command.C\_POP;

        } else if(commands[0].equals("push")) {

            commandType = Command.C\_PUSH;

        }

    }

    /\*\*

     \* Parses the first argument of the vm command line

     \*/

    private void parseArg1() {

        switch(commandType) {

            case NO\_COMMAND:

                arg1 = "";

                break;

            case C\_ARITHMETIC:

                arg1 = commands[0];

                break;

            default:

                arg1 = commands[1];

                break;

        }

    }

    /\*\*

     \* Parses the second argument of the vm command line

     \* Should only be called if command type is push or pop

     \*/

    private void parseArg2() {

        arg2 = Integer.parseInt(commands[2]);

    }

    /\*\*

     \* Return the Command enum for the command type of the current line

     \* @return Command type enum

     \*/

    public Command getCommandType() {

        return commandType;

    }

    /\*\*

     \* Return the line number of the current line

     \* @return The line number of current line

     \*/

    public int getLineNumber() {

        return lineNumber;

    }

    /\*\*

     \* Return the first argument of the vm command line, if there is one. May

     \* have already been initialized

     \* @return the first argument of the vm command line

     \*/

    public String getArg1() {

        return arg1;

    }

    /\*\*

     \* Return the second argument of the vm command line, if there is one. May

     \* have already been initialized

     \* @return the second argument of the vm command

     \*/

    public int getArg2() {

        return arg2;

    }

}

CodeWriter.java

/\*\*

 \* @author  Raul Aguilar

 \* @date    October 26, 2019

 \* CodeWriter: Translates VM commands into Hack assembly code

 \*/

import java.io.FileNotFoundException;

import java.io.PrintWriter;

public class CodeWriter {

    PrintWriter outputFile = null;

    private int labelCounter = 1;

    private String file = "";

    /\*\*

     \* Opens the output file and gets ready to write into it

     \* @param fileName Name of the output file

     \*/

    public void CodeWriter(String fileName) {

        try {

            outputFile = new PrintWriter(fileName);

            file = fileName.substring(0, fileName.lastIndexOf('.'));

        } catch (FileNotFoundException e) {

            System.err.println("Could not open output file " + fileName);

            System.err.println("Run program again, make sure you have write permissions, etc.");

            System.err.println("Program exiting.");

            System.exit(0);

        }

    }

    /\*\*

     \* Closes the output file

     \*/

    public void close() {

        outputFile.close();

    }

    /\*\*

     \* Writes an infinite loop to prevent NOP slide and the end of translation

     \*/

    public void writeInfiniteLoop() {

        outputFile.println("(END)");

        outputFile.println("@END");

        outputFile.println("0;JMP");

    }

    /\*\*

     \* Writes the assembly code for the given arithmetic command

     \* @param command   The arithmetic command to perform

     \*/

    public void writeArithmetic(String command) {

        switch(command) {

            case "add": case "sub":

                writeAddSub(command);

                break;

            case "neg": case "not":

                writeNegateNot(command);

                break;

            case "eq": case "lt": case "gt":

                writeEqualities(command);

                break;

            case "and": case "or":

                writeAndOr(command);

                break;

        }

    }

    /\*\*

     \* Writes the assembly code that is the translation of the given command,

     \* where command is either C\_PUSH or C\_POP

     \* @param command   Push or Pop command

     \* @param segment   Memory segment to manipulate

     \* @param index     Memory address to go to

     \*/

    public void writePushPop(Command command, String segment, int index) {

        String seg = "";

        switch(segment) {

            case "local":

                seg = "LCL";

                break;

            case "argument":

                seg = "ARG";

                break;

            case "this": case "that":

                seg = segment.toUpperCase();

                break;

            case "pointer":

                if(index == 0) {

                    segment = "THIS";

                    break;

                }

                if(index == 1) {

                    segment = "THAT";

                    break;

                }

            case "static":

                seg = file;

                break;

        }

        if(command == Command.C\_PUSH) {

            if(segment.equals("constant")) {

                writePushCont(index);

            } else if(segment.equals("THIS") || segment.equals("THAT")) {

                writePushPointer(segment);

            } else if(segment.equals("temp")) {

                writePushTemp(index);

            } else if(seg.equals(file)) {

                writePushStatic(file, index);

            } else {

                writePush(seg, index);

            }

        }

        if(command == Command.C\_POP) {

            if(segment.equals("THIS") || segment.equals("THAT") ) {

                writePopPointer(segment);

            } else if(segment.equals("temp")) {

                writePopTemp(index);

            } else if(seg.equals(file)) {

                writePopStatic(file, index);

            } else {

                writePop(seg, index);

            }

        }

    }

    /\* WRITE ARITHMETIC AND LOGICAL COMMANDS \*/

    /\*\*

     \* Write to file assembly code for 'add' and 'sub' arithmetic

     \* depending on the given command

     \* @param command   The arithmetic command to perform

     \*/

    private void writeAddSub(String command) {

        outputFile.println("@SP");

        outputFile.println("AM = M - 1");

        outputFile.println("D = M");

        outputFile.println("A = A - 1");

        if(command.equals("add")) {

            outputFile.println("M = M + D");

        } else if(command.equals("sub")) {

            outputFile.println("M = M - D");

        }

    }

    /\*\*

     \* Write to file assembly code for negate or not arithmetic command

     \*/

    private void writeNegateNot(String command) {

        outputFile.println("@SP");

        outputFile.println("A = M - 1");

        if(command.equals("neg")) {

            outputFile.println("M = -M");

        } else if(command.equals("not")) {

            outputFile.println("M = !M");

        }

    }

    /\*\*

     \* Write to file assembly code for 'and' and 'or' depending on

     \* the given arithmetic command

     \* @param command   The arithmetic command to perform

     \*/

    private void writeAndOr(String command) {

        outputFile.println("@SP");

        outputFile.println("AM = M - 1");

        outputFile.println("D = M");

        outputFile.println("A = A - 1");

        if(command.equals("and")) {

            outputFile.println("M = D&M");

        } else if (command.equals("or")) {

            outputFile.println("M = D|M");

        }

    }

    /\*\*

     \* Write to file the assembly code for equality comparison

     \* @param command The equality command

     \*/

    private void writeEqualities(String command) {

        if(command.equals("lt")) {

            command = "gt";

        } else if(command.equals("gt")) {

            command = "lt";

        }

        outputFile.println("@SP");

        outputFile.println("AM = M - 1");

        outputFile.println("D = M");

        outputFile.println("A = A - 1");

        outputFile.println("D = D - M");

        outputFile.println("@\_"+ labelCounter++);

        outputFile.println("D;J" + command.toUpperCase());

        outputFile.println("@\_"+ labelCounter++);

        outputFile.println("D = 0");

        outputFile.println("0;JMP");

        outputFile.println("(\_" + (labelCounter-2) + ")");

        outputFile.println("D = -1");

        outputFile.println("(\_" + (labelCounter-1) + ")");

        outputFile.println("@SP");

        outputFile.println("A = M - 1");

        outputFile.println("M = D");

    }

    /\* WRITE TO MEMORY SEGMENTS \*/

    /\*\*

     \* Helper method to push a value to stack

     \* Moves SP forward by 1 and stores value in previous location

     \*/

    private void writePushD() {

        outputFile.println("@SP");

        outputFile.println("M = M + 1");

        outputFile.println("A = M - 1");

        outputFile.println("M = D");

    }

    /\*\*

     \* Helper method for pop commands

     \* Moves the SP back by 1 and stores the value

     \*/

    private void writePopD() {

        outputFile.println("@SP");

        outputFile.println("AM = M - 1");

        outputFile.println("D = M");

    }

    /\*\*

     \* Writes assembly code for the vm 'pop' command

     \* Takes the memory segment from the vm code and translates it into its

     \* corresponding predefined symbol

     \* Uses the index to find the correct memory address

     \* @param segment   Memory segment to pop to

     \* @param index     Index of the memory segment address

     \*/

    private void writePop(String seg, int index) {

        // write to file

        writePopD();

        outputFile.println("@"+seg);

        if(index > 0) {

            outputFile.println("A = M + 1");

            for(int i = 1; i < index; i++) {

                outputFile.println("A = A + 1");

            }

        } else {

            outputFile.println("A = M");

        }

        outputFile.println("M = D");

    }

    /\*\*

     \* Writes assembly code to pop to THIS or THAT ram locations

     \* @param segment   THIS or THAT depending on given pointer index

     \*/

    private void writePopPointer(String segment) {

        writePopD();

        outputFile.println("@"+segment);

        outputFile.println("M = D");

    }

    /\*\*

     \* Writes assembly code to pop into static memory segment

     \*/

    private void writePopStatic(String file, int index) {

        writePopD();

        outputFile.println("@"+file+"."+index);

        outputFile.println("M = D");

    }

    /\*\*

     \* Writes assembly code to pop into the temp memory segment

     \* @param index

     \*/

    private void writePopTemp(int index) {

        writePopD();

        outputFile.println("@"+(5+index));

        outputFile.println("M = D");

    }

    /\*\*

     \* Helper method to write assembly code for push constants

     \* @param index RAM location / constant

     \*/

    private void writePushCont(int index) {

        if(index > 1) {

            outputFile.println("@"+index);

            outputFile.println("D = A");

            writePushD();

        } else if(index == 1) {

            outputFile.println("@SP");

            outputFile.println("M = M + 1");

            outputFile.println("A = M - 1");

            outputFile.println("M = 1");

        } else {

            System.out.println("Index is not a positive number.");

        }

    }

    /\*\*

     \* Pushes data to the stack from the temp memory segment

     \* @param index Index of the memory location to access

     \*/

    private void writePushTemp(int index) {

        outputFile.println("@"+(5+index));

        outputFile.println("D = M");

        writePushD();

    }

    /\*\*

     \* Pushes data to stack from static memory

     \* @param file  Name of the file

     \* @param index Index of memory location to access

     \*/

    private void writePushStatic(String file, int index) {

        outputFile.println("@"+file+'.'+index);

        outputFile.println("D = M");

        writePushD();

    }

    /\*\*

     \* Pushes from THIS or THAT ram locations depending on given pointer index

     \* @param segment   THIS or THAT

     \*/

    private void writePushPointer(String segment) {

        outputFile.println("@"+segment);

        outputFile.println("D = M");

        writePushD();

    }

    /\*\*

     \* Writes assembly code to push from memory segment to the stack

     \* @param seg   Memory segment of stack

     \* @param index Location in the memory segment

     \*/

    private void writePush(String seg, int index) {

        // Get data from segment and index

        outputFile.println("@"+seg);

        if(index > 0) {

            outputFile.println("A = M + 1");

            for(int i = 1; i < index; i++) {

                outputFile.println("A = A + 1");

            }

        } else {

            outputFile.println("A = M");

        }

        outputFile.println("D = M");

        // push it to the stack

        writePushD();

    }

}

Command.java

public enum Command {

    NO\_COMMAND, C\_ARITHMETIC, C\_PUSH, C\_POP, C\_LABEL, C\_GOTO, C\_IF,

    C\_FUNCTION, C\_RETURN, C\_CALL

}

Basic Test

A screenshot of a cell phone

Description automatically generated

Pointer Test

A screenshot of a cell phone

Description automatically generated

Simple Add

A screenshot of a cell phone

Description automatically generated

Stack Test

A screenshot of a cell phone

Description automatically generated

Static Test

A screenshot of a cell phone

Description automatically generated