Raul Aguilar

CS 220 2367

Homework 7 Assembler

Assembler.java

/\*\*

 \* @author Raul Aguilar

 \* @date    October 16, 2019

 \*/

import java.io.FileOutputStream;

import java.io.FileNotFoundException;

import java.util.Scanner;

import java.io.PrintWriter;

public class Assembler {

    // ALGORITHM:

    // get input file name

    // create output file name and stream

    // create symbol table

    // do first pass to build symbol table (no output yet)

    // do second pass to output translated ASM to HACK code

    // print "done" message to user

    // close output file stream

    public static void main(String[] args) {

        String inputFileName, outputFileName;

        PrintWriter outputFile = null;

        SymbolTable symbolTable = new SymbolTable();

        int romAddress = 0, ramAddress = 16;

        // get input file name from command line or console input

        if(args.length == 1) {

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

            inputFileName = args[0];

        } else {

            Scanner keyboard = new Scanner(System.in);

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

            System.out.println("Don't forget the .asm etension: ");

            inputFileName = keyboard.nextLine();

            keyboard.close();

        }

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

        try {

            outputFile = new PrintWriter(new FileOutputStream(outputFileName));

        } catch(FileNotFoundException ex) {

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

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

            System.exit(0);

        }

        firstPass(inputFileName, symbolTable, romAddress, ramAddress);

        secondPass(inputFileName, symbolTable, outputFile, romAddress, ramAddress);

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

        outputFile.close();

        System.exit(0);

    }

    /\*\*

     \* The first pass through the file finds and stores user-defined variables and labels in the symbol

     \* table without writing anything to the output file

     \* @param inputFileName the file being read

     \* @param symbolTable   the symbol table to store variable and lables (initialzies with predefined symbols)

     \* @param romAddress    the current PC rom address of the instruction

     \* @param ramAddress    the current ram address to store the variable in

     \* @return  symbol table filled-in with variables and labels found in file

     \*/

    private static SymbolTable firstPass(String inputFileName, SymbolTable symbolTable, int romAddress, int ramAddress) {

        Parser p = new Parser();

        symbolTable.SymbolTable();

        p.Parser(inputFileName);

        while(p.hasMoreCommands()) {

            p.advance();

            if(p.getCommandType() == Command.L\_COMMAND) {

                symbolTable.addEntry(p.getSymbol(), romAddress, p.getLineNumber());

            }

            if(p.getCommandType() == Command.A\_COMMAND) {

                try {

                    int decimal = Integer.parseInt(p.getSymbol());

                } catch(NumberFormatException notADecimal) {

                    if(!symbolTable.contains(p.getSymbol()) && Character.isLowerCase(p.getSymbol().charAt(0)) ) {

                        symbolTable.addEntry(p.getSymbol(), ramAddress, p.getLineNumber());

                        ramAddress++;

                    }

                }

                romAddress++;

            }

            if(p.getCommandType() == Command.C\_COMMAND) {

                romAddress++;

            }

        }

        return symbolTable;

    }

    /\*\*

     \* Second pass through the file converts each line to binary code, while using the filled-in symbol

     \* table from the first pass to convert symbols and labls

     \* @param inputFileName the file being read

     \* @param symbolTable   the predefined symbol table

     \* @param outputFile    the name of the output HACK file

     \* @param romAddress    the current PC rom address of the instruction

     \* @param ramAddress    the current ram address for user-defined variables

     \*/

    private static void secondPass(String inputFileName, SymbolTable symbolTable, PrintWriter outputFile, int romAddress, int ramAddress) {

        Parser p = new Parser();

        p.Parser(inputFileName);

        while(p.hasMoreCommands()) {

            p.advance();

            if(p.getCommandType() == Command.C\_COMMAND) {

                String instruction = "111" + p.getComp() + p.getDest() + p.getJump() + '\n';

                outputFile.write(instruction);

                romAddress++;

            }

            if(p.getCommandType() == Command.A\_COMMAND) {

                try {

                    int decimal = Integer.parseInt(p.getSymbol());

                    String dec = Code.decimalToBinary(decimal) + '\n';

                    outputFile.write(dec);

                    romAddress++;

                } catch(NumberFormatException notADecimal) {

                    if(symbolTable.contains(p.getSymbol())) {

                        String dec = Code.decimalToBinary(symbolTable.getAddress(p.getSymbol())) + '\n';

                        outputFile.write(dec);

                    } else {

                        symbolTable.addEntry(p.getSymbol(), ramAddress, p.getLineNumber());

                    }

                    romAddress++;

                }

            }

        }

    }

}

Code.java

/\*\*

 \* @author Raul Aguilar

 \* @date    October 14, 2019

 \*/

import java.util.HashMap;

public class Code {

    private HashMap<String, String> compCodes = new HashMap<String, String>();

    private HashMap<String, String> destCodes = new HashMap<String, String>();

    private HashMap<String, String> jumpCodes = new HashMap<String, String>();

    /\*\*

     \* Initializes hashmaps with binary codes for easy lookup

     \*/

    public void Code() {

        // Comp codes

        compCodes.put("0", "0101010");

        compCodes.put("1", "0111111");

        compCodes.put("-1", "0111010");

        compCodes.put("D", "0001100");

        compCodes.put("A", "0110000");

        compCodes.put("M", "1110000");

        compCodes.put("!D", "0001101");

        compCodes.put("!A", "0110001");

        compCodes.put("!M", "1110001");

        compCodes.put("-D", "0001111");

        compCodes.put("-A", "0110011");

        compCodes.put("D+1", "0011111");

        compCodes.put("1+D", "0011111");

        compCodes.put("A+1", "0110111");

        compCodes.put("1+A", "0110111");

        compCodes.put("M+1", "1110111");

        compCodes.put("1+M", "1110111");

        compCodes.put("D-1", "0001110");

        compCodes.put("-1+D", "0001110");

        compCodes.put("A-1", "0110010");

        compCodes.put("-1+A", "0110010");

        compCodes.put("M-1", "1110010");

        compCodes.put("-1+M", "1110010");

        compCodes.put("D+A", "0000010");

        compCodes.put("A+D", "0000010");

        compCodes.put("D+M", "1000010");

        compCodes.put("M+D", "1000010");

        compCodes.put("D-A", "0010011");

        compCodes.put("D-M", "1010011");

        compCodes.put("A-D", "0000111");

        compCodes.put("M-D", "1000111");

        compCodes.put("D&A", "0000000");

        compCodes.put("D&M", "1000000");

        compCodes.put("D|A", "0010101");

        compCodes.put("D|M", "1010101");

        // Dest codes

        destCodes.put(null, "000");

        destCodes.put("", "000");

        destCodes.put("\"null\"", "000");

        destCodes.put("M", "001");

        destCodes.put("D", "010");

        destCodes.put("MD", "011");

        destCodes.put("DM", "011");

        destCodes.put("A", "100");

        destCodes.put("AM", "101");

        destCodes.put("MA", "101");

        destCodes.put("AD", "110");

        destCodes.put("DA", "110");

        destCodes.put("AMD", "111");

        destCodes.put("ADM", "111");

        destCodes.put("MAD", "111");

        destCodes.put("MDA", "111");

        destCodes.put("DAM", "111");

        destCodes.put("DMA", "111");

        // Jump codes

        jumpCodes.put(null, "000");

        jumpCodes.put("", "000");

        jumpCodes.put("\"null\"", "000");

        jumpCodes.put("JGT", "001");

        jumpCodes.put("JEQ", "010");

        jumpCodes.put("JGE", "011");

        jumpCodes.put("JLT", "100");

        jumpCodes.put("JNE", "101");

        jumpCodes.put("JLE", "110");

        jumpCodes.put("JMP", "111");

    }

    /\*\*

     \* Returns binary code for given comp mnemonic

     \* @param mnemonic the key given

     \* @return 7 bits for comp key

     \*/

    public String getComp(String mnemonic) {

        return compCodes.get(mnemonic);

    }

    /\*\*

     \* Returns binary code for dest mnemonic

     \* @param mnemonic the key given

     \* @return 3 bits for dest key

     \*/

    public String getDest(String mnemonic) {

        return destCodes.get(mnemonic);

    }

    /\*\*

     \* Returns binary code for jump mnemonic

     \* @param mnemonic the key given

     \* @return 3 bits for jump key

     \*/

    public String getJump(String mnemonic) {

        return jumpCodes.get(mnemonic);

    }

    /\*\*

     \* Converts a decimal number to binary

     \* @param n decimal number

     \* @return  binary representation of decimal number

     \*/

    public static String decimalToBinary(int n) {

        String binary = "";

        do {

          binary = (n%2) + binary;

          n /= 2;

        } while(n > 0);

        while(binary.length() < 16) {

            binary = "0" + binary;

        }

        return binary;

      }

}

Parser.java

/\*\*

 \* @author  Raul Aguilar

 \* @date    October 16, 2019

 \*/

 import java.io.\*;

 import java.util.Scanner;

 public class Parser {

    private Scanner inputFile;

    private int lineNumber;

    private String rawLine;

    private String cleanLine;

    private Command commandType;

    private String symbol;

    private String destMnemonic;

    private String compMnemonic;

    private String jumpMnemonic;

    private Code c = new Code();

    Command command;

    /\*\*

     \* Opens input file and prepares to parse

     \* If file cannot be found ends program with error message

     \* @param inFileName

     \*/

    public void Parser(String inFileName) {

        try {

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

        } catch (FileNotFoundException e) {

            System.out.println("File could not be found. Ending program.");

            System.exit(0);

        }

    }

    /\*\*

     \* Returns boolean if more commands left, closes stream if not

     \* @return True if more commands, else 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 only be called if hasMoreCommands() is true.

     \* Initially there is no current command.

     \*/

    public void advance() {

        lineNumber++;

        rawLine = inputFile.nextLine();

        cleanLine();

        parseCommandType();

        parse();

    }

    /\*\*

     \* Reads raw line from file and strips it of whitespace

     \* and comments

     \*/

    private void cleanLine() {

        int commentIndex;

        if(rawLine == null) {

            cleanLine = "";

        } else {

            // remove whitespace

            cleanLine = rawLine.replaceAll(" ", "");

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

            //remove comments

            commentIndex = cleanLine.indexOf("/");

            if(commentIndex != -1) {

                cleanLine = cleanLine.substring(0, commentIndex);

            }

        }

    }

    /\*\*

     \* Guesses which command type it is from clean line

     \*/

    private void parseCommandType() {

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

            commandType = command.NO\_COMMAND;

        } else {

            char first = cleanLine.charAt(0);

            if(first == '(') {

                commandType = Command.L\_COMMAND;

            } else if(first == '@') {

                commandType = Command.A\_COMMAND;

            } else {

                commandType = Command.C\_COMMAND;

            }

        }

    }

    /\*\*

     \* Helper method: parses line depending on instuction type

     \* Appropriate parts of instruction filled

     \*/

    private void parse() {

        if(commandType == Command.L\_COMMAND || commandType == Command.A\_COMMAND) {

            parseSymbol();

        } else if(commandType == Command.C\_COMMAND) {

            parseComp();

            parseDest();

            parseJump();

        }

    }

    /\*\*

     \* Parses symbol from A- or L-Commands

     \*/

    private void parseSymbol() {

        if(commandType == Command.L\_COMMAND) {

            int begin = cleanLine.indexOf('(');

            int end = cleanLine.indexOf(')');

            symbol = cleanLine.substring(begin+1, end);

        }

        if(commandType == Command.A\_COMMAND) {

            symbol = cleanLine.substring(1);

        }

    }

    /\*\*

     \* Helper method: parses line to get dest part

     \*/

    private void parseDest() {

        c.Code();

        int equals = cleanLine.indexOf('=');

        if(equals != -1) {

            destMnemonic = cleanLine.substring(0, equals);

            destMnemonic = c.getDest(destMnemonic);

        } else {

            destMnemonic = null;

            destMnemonic = c.getDest(destMnemonic);

        }

    }

    /\*\*

     \* Helper method: parses line to get comp part

     \*/

    private void parseComp() {

        c.Code();

        int equals = cleanLine.indexOf('=');

        int semicolon = cleanLine.indexOf(';');

        if(semicolon == -1 && equals > 0) {

            compMnemonic = cleanLine.substring(equals+1);

            compMnemonic = c.getComp(compMnemonic);

        } else if(semicolon > 0 && equals > 0) {

            compMnemonic = cleanLine.substring(equals+1, semicolon);

            compMnemonic = c.getComp(compMnemonic);

        } else if(semicolon > 0 && equals == -1) {

            compMnemonic = cleanLine.substring(0, semicolon);

            compMnemonic = c.getComp(compMnemonic);

        }

    }

    /\*\*

     \* Helper method: parses line to get jump part

     \*/

    private void parseJump() {

        c.Code();

        int semicolon = cleanLine.indexOf(';');

        if(semicolon != -1) {

            jumpMnemonic = cleanLine.substring(semicolon+1);

            jumpMnemonic = c.getJump(jumpMnemonic);

        } else {

            jumpMnemonic = null;

            jumpMnemonic = c.getJump(jumpMnemonic);

        }

    }

    /\*\*

     \* Getter for the command type of the current line

     \* @return Command enum for command type

     \*/

    public Command getCommandType() {

        return commandType;

    }

    /\*\*

     \* Getter for the symbol parsed from the line

     \* @return String for symbol

     \*/

    public String getSymbol() {

        return symbol;

    }

    /\*\*

     \* Getter for dest part of C-instuctruction

     \* May be empty

     \* @return the dest mnemonic

     \*/

    public String getDest() {

        return destMnemonic;

    }

    /\*\*

     \* Getter for the comp part of the C-instuction

     \* @return the comp mnemonic

     \*/

    public String getComp() {

        return compMnemonic;

    }

    /\*\*

     \* Getter for the jump part of the C-instuction

     \* May be empty

     \* @return the jump mnemonic

     \*/

    public String getJump() {

        return jumpMnemonic;

    }

    /\*\*

     \* Get the current line from the file

     \* @return line from the file

     \*/

    public String getRawLine() {

        return rawLine;

    }

    /\*\*

     \* Get the clean version of the raw line

     \* @return cleaned up line

     \*/

    public String getCleanLine() {

        return cleanLine;

    }

    /\*\*

     \* Get the line number of the symbol encountered

     \* @return current line number

     \*/

    public int getLineNumber() {

        return lineNumber;

    }

 }

SymbolTable.java

/\*\*

 \* @author Raul Aguilar

 \* @date    October 14, 2019

 \*/

import java.util.HashMap;

public class SymbolTable {

    private static String ALL\_VALID\_CHARS = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789\_.$:";

    private HashMap<String, Integer> symbolTable = new HashMap<>();

    /\*\*

     \* Initializes hashmap with predefined symbols

     \*/

    public void SymbolTable() {

        for(int i = 0; i < 16; i++) {

            symbolTable.put("R"+i, i);

        }

        symbolTable.put("SP", 0);

        symbolTable.put("LCL", 1);

        symbolTable.put("ARG", 2);

        symbolTable.put("THIS", 3);

        symbolTable.put("THAT", 4);

        symbolTable.put("SCREEN", 16384);

        symbolTable.put("KBD", 24576);

    }

    /\*\*

     \* Adds new pair of symbol/address to hasmap

     \* @param symbol    name of symbol to add

     \* @param address   address associated with that symbol

     \* @param lineNumber line number where symbol encountered

     \* @return true if pair is added, false if illegal name

     \*/

    public boolean addEntry(String symbol, int address, int lineNumber) {

        boolean entryAdded = false;

        if(contains(symbol)) {

            entryAdded = false;

        }

        if(isValidSymbolName(symbol, lineNumber)) {

            symbolTable.put(symbol, address);

            entryAdded = true;

        }

        return entryAdded;

    }

    /\*\*

     \* Returns boolean of whether hashmap has symbol or not

     \* @param symbol symbol to check

     \* @return true if symbol exist, false if not

     \*/

    public boolean contains(String symbol) {

        if(symbolTable.containsKey(symbol)) {

            return true;

        } else {

            return false;

        }

    }

    /\*\*

     \* Returns address in hashmap of given symbol

     \* PRECONDITION: symbol is in hashmap(check w/ contains())

     \* @param symbol to obtain address

     \* @return address associated with symbol in hashmap

     \*/

    public int getAddress(String symbol) {

        return symbolTable.get(symbol);

    }

    /\*\*

     \* Boolean to check if user-defined symbol name is valid

     \* @param symbol the symbol being tested

     \* @param lineNumber line number the symbol is found on

     \* @return true if symbol name is valid, otherwise exit with error message

     \*/

    private static boolean isValidSymbolName(String symbol, int lineNumber) {

        boolean isValidName = false;

        for(char c:symbol.toCharArray()) {

            if(ALL\_VALID\_CHARS.indexOf(c) == -1) {

                System.out.printf("Symbol name is not valid on line %d. Program exiting.", lineNumber);

                isValidName = false;

                System.exit(0);

            } else {

                isValidName = true;

            }

        }

        return isValidName;

    }

 }

A screenshot of a social media post

Description automatically generated