Lab 3 – Linker/Loader

Modified Assembler Programmer's Guide

CSE 3903

Spring 2023

Group: Worst Name Ever

Sade Ahmed

Jeremy Bogen

Mani Kamali

Giridhar Srikanth

Date of Submission: 04/19/2023

Table of Contents

Introduction

This document acts as a guide to the WNE-assembler. This includes descriptions of each component as well as guides to our design philosophy and data structures, and assumes a high level of familiarity with the Java programming language and the principles of OOP. For any reference regarding how to use the program, please see the WNE-assembler User's Guide

Data Structures

Machine Operation Table (MOT)

The machine Operation Table, abbreviated to MOT throughout this document, stores information about the assembly language's machine operations (for an overview of the machine operations please refer to the user's guide). The machine operation table is declared and defined in the "Machine_Op_Table.java" class. The main data structure is defined as so:

```
/**
 * Map to represent the machine op table.
 */
private HashMap<String, Machine_Op_Info> table;
```

Machine_Op_Info is explained in its own section, but, in simple terms, for each machine operation it stores the instruction's mnemonic, opcode, size, and format.

Instantiating the class initializes the hashmap and calls loadTable(), which populates the table, here is an excerpt to demonstrate how loadTable() works:

```
public void loadTable() {
        // Data processing instructions
        this.table.put("ADD", new Machine_Op_Info(Machine_Op_Ins_Enum.ADD, ADD_OPCODE, ONE, Machine_Op_Format.NONE));
        this.table.put("AND", new Machine_Op_Info(Machine_Op_Ins_Enum.AND, AND_OPCODE, ONE, Machine_Op_Format.NONE));
        this.table.put("NOT", new Machine_Op_Info(Machine_Op_Ins_Enum.NOT, NOT_OPCODE, ONE, Machine_Op_Format.NONE));
```

To see the full method, check out "Machine_Op_Table.java" in the code appendices.

With the table initialized, the MOT can be used to query information about Machine Operations using their mnemonic strings.

Machine Operation Info

The machine operation table stores a variety of information about each instruction. The data format for this information is declared and defined in the "Machine_Op_Info.java" class. It, at the time of writing, stores four values: the instruction mnemonic, the given opcode, the size, and the format of the instruction:

```
/*
 * Enumeration to store the mnemonic name of the instruction.
 */
private Machine_Op_Ins_Enum ins;
```

```
/*
 * Value for the given opcode.
 */
private int opcode;
```

```
/*
 * Length of the instruction.
 */
private int size;
```

```
/*
 * Enumeration to store the type of format (page offset, index offset, or none).
 */
private Machine_Op_Format format;
```

The Machine_Op_Info class initializes all these values when constructed:

```
public Machine_Op_Info(Machine_Op_Ins_Enum ins, int opcode, int size, Machine_Op_Format format)
        this.ins = ins;
        this.opcode = opcode;
        this.size = size;
        this.format = format;
}
```

Once initialized, these values should not change, so all values are private, and have getters, but no setters. Here they are for reference:

```
public Machine_Op_Ins_Enum getInstructionName() {
        return this.ins;
}
```

```java
public int getOpcode() {
        return this.opcode;
}
```

```java
public int getSize() {
        return this.size;
}
```

```java
public Machine_Op_Format getFormat() {
        return this.format;
}
```

Pseudo Operation Table (POT)

The Pseudo-Ops table class serves a similar purpose to the MOT where the class stores each of the names and other required parameters into a Hashmap. The class also initializes a String Parser object (more details about this class are mentioned in the String Parser section) that allows for parsing constant values for opcodes that require variable length sizes of memory to be allocated. The private global members of the POT can be seen below:

```java
/**
 * Map to represent the machine op table.
 */
private HashMap<String, Pseudo_Op_Info> table;

/**
 * Parser specialized to deal with parsing immediates in the assembly language.
 */
private String_Parser sp;
```

The constructor for the POT instantiates a new HashMap object, which is the concrete representation of the POT, the string parser, and calls a method to fill the “table” with the instruction name and its value having the format, sizes, and instruction name (as an enum).

```
public Pseudo_Op_Table() {
    this.table = new HashMap<>();
    sp = new String_Parser();
    loadTable();
}
```

The loadTable() method loads every possible Pseudo operation into the table with the info mentioned previously (more about the information section in Pseudo Operation Info). A sample of how the input is loaded can be seen below:

```
public void loadTable() {
    // Ops with size 0
    this.table.put(".ORIG", new Pseudo_Op_Info(Pseudo_Op_Ins_Enum.ORIG, ZERO, Pseudo_Op_Format.DEFINITE));
    this.table.put(".EQU", new Pseudo_Op_Info(Pseudo_Op_Ins_Enum.EQU, ZERO, Pseudo_Op_Format.DEFINITE));
```

There are other getter methods for getting components of the table such as the format and length in the POT class. Two other methods get the length of the block (**.**BLKW operation) based on the operand passed. The method attempts to parse the operand and returns the block of memory to allocate, otherwise, it will return an invalid value that is later checked in the calling class to throw an exception.

Pseudo Operation Info

The pseudo operation table stores a variety of information about each instruction. The data format for this information is declared and defined in the “Pseudo_Op_Info.java” class. It stores the following fields: an enumerated version of the instruction name, the length of the instruction, and the format for the type of length.

```
/*
 * Enumeration to store the mnemonic name of the Pseudo-op.
 */
private Pseudo_Op_Ins_Enum ins;

/*
 * Length of the instruction.
 */
private int length;

/*
 * Enumeration to store the type of format (page offset, index offset, or none).
 */
private Pseudo_Op_Format format;
```

The methods for this class are mainly getter methods as all the values are set in the Pseudo_Ops_Table.java class. An example of one of the three getters can be seen below.

```
public Pseudo_Op_Ins_Enum getInstructionName() {
    return this.ins;
}
```

This class is mainly used to represent the “value” portion of the Pseudo Operations Table as the concrete representation of the table is a HashMap.

Symbol Table

The symbol table, part of the “intermediate file” in between passes 1 and 2, is represented as a HashMap. The key is a String that represents the name of the symbol, and the value(s) is an array of Strings. Each value array has a length of two, the first String representing the value of the symbol and the second representing whether the symbol is relative or absolute.

To fill the symbol table, the ArrayList of Lists that is formed through the file validation is iterated through. Each iteration checks if the current line has a symbol, and if it does, fills the symbol table accordingly with the value of the location counter (or the value of the operand if the operation is a .EQU) and an “R” or “A”, depending on whether the symbol is relative or absolute.

Literal Table

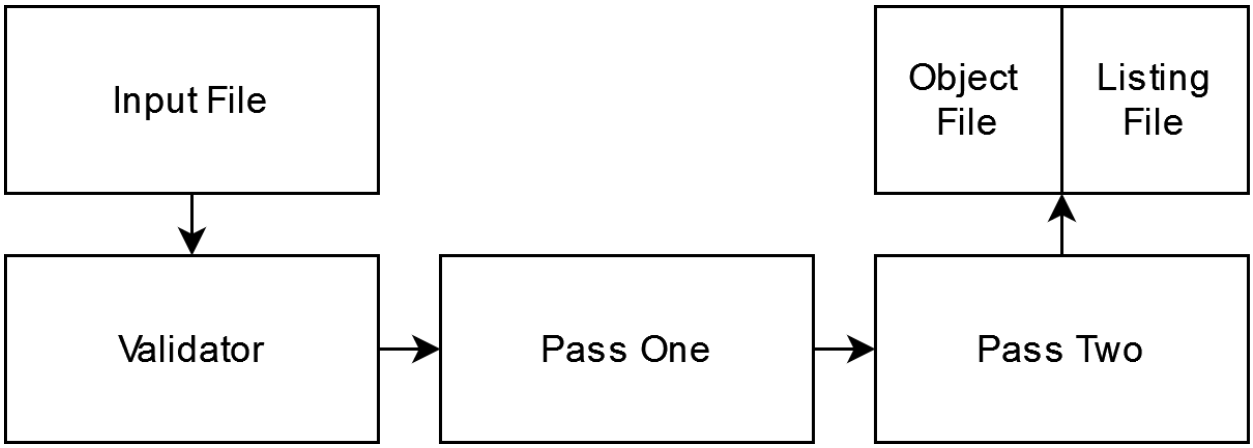
The literal table, the other half of the “intermediate file” in between passes 1 and 2, is represented as a HashMap. The key is a String that represents the value of the literal, and the value is a String that represents the spot in memory that will be filled by the literal.

During the initial pass through the ArrayList of Lists, if the program sees a line with a literal in it, it will pull out said literal into a separate ArrayList of Strings to be stored until the initial pass has been completed. Once this pass is done, the program will iterate through this separate ArrayList, filling the literal table accordingly with the value of the literal and the location at which it is to be stored.

The reason why this iteration is done after the initial pass is completed is that the location at which all literals are stored is after all instructions have been stored, so the program must know where the location of the final instruction before assigning the locations that come after it to any literals that are in the program.

Data and Information Flow

The WNE Assembler has a fairly simple data flow. At its most abstract it looks something like this:



Note the entire file is processed by the validator before being passed on. This makes the compilation of a valid file slightly slower, but makes the reporting of issues in invalid files faster, which is better for development time.

This dataflow logic is expressed in the program's main method, which can be found in App.java. The program starts by reading in a file, then iterating through each line and adding them to an ArrayList of ArrayLists called lines.

```
BufferedReader reader = new BufferedReader(asmFile);
reader.lines().forEach((String line) -> {
        try {
                tempList = validator.validate(line);
                if (tempList != null) {
                        lines.add(tempList);
                }
        } catch (Exception e) {
                throw new RuntimeException(e);
        }
});
```

The interior ArrayList is an ArrayList of Strings, each containing 4 elements. These elements are:

- Symbol
    - If there is no symbol for the line, the symbol will be a blank string
- Operation/OpCode
- Operand(s)
- Line number

In order for the validator to split these lines up, it splits the line based on the given Syntax specifications in the documentation.

1-6	Label, if any, left justified
7-9	Unused (i.e., white space)
10-14	Operation field
15-17	Unused (i.e., white space)
18-end of record	Operands and comments (comments begin with a semicolon (;)

The validator also strips any comments out by searching for the first instance of a semicolon, as our assembler does not output any comments in either of the files so they are unnecessary to the program.

Using the Strings that contain the label, operation, and operands, and after trimming any white space, the validator uses the clauses defined in the User's Guide on pages 10 and 11 to confirm whether or not the program is valid.

Once the validator has confirmed that the program is indeed a valid file that can be turned into the output file and the listing file, the Pass One class will then take in the ArrayList of Lists that the Validator creates.

Using this ArrayList, pass one will do two main things: create the symbol table, and create the literal table

```
loc = fillSymTable(arr, SymTable, loc);
loc = fillLitTable(LitTable, loc, LitArray);
return loc;
```

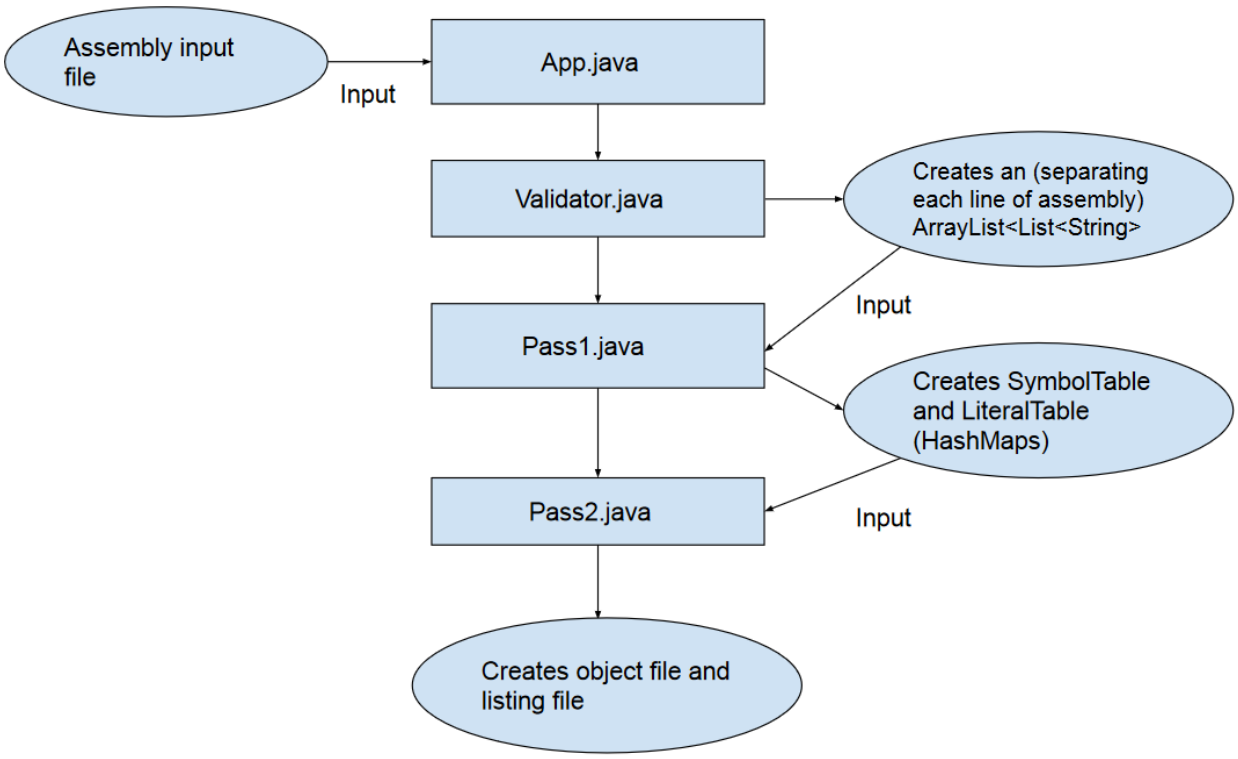
To fill the symbol table, the program iterates through every line of assembly code (not including the comment lines), which each is represented by a list as explained before. Pass one will take the symbol, operation, and operand from the List, and then check if there is a symbol at all in this line (and make sure that it is not a duplicate symbol). If there is not, it will continue to the next line.

If there is, however, it will then fill the symbol table with the corresponding information. To do this, it first attempts to fill the value array for the symbol. The program checks the instruction and operand to determine whether or not the symbol is relative or absolute, using a series of methods that see if the operand is an absolute value. Since the only way to get an absolute symbol is through a .EQU instruction, this process is streamlined a bit.

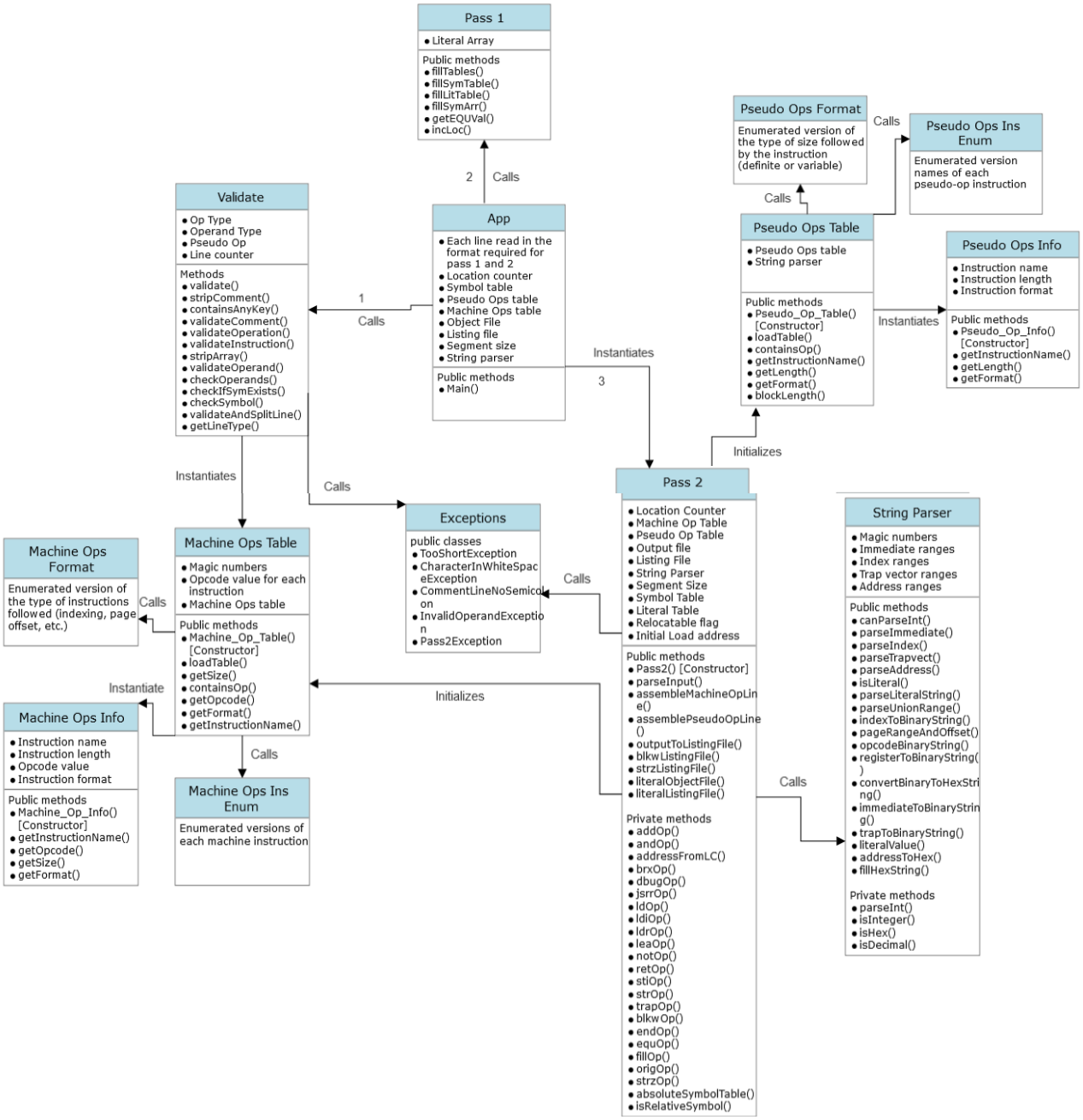
Once the symbol table is filled, it then iterates through the array of literals that were taken out during the process of filling the symbol table, putting them into the LitTable HashMap, taking the new value of the location counter and putting those corresponding values with the literals, incrementing the location counter by one with each literal. With the two tables being filled out, pass one then gives the filled symbol table and literal table back to the main program, where the two HashMaps, along with the full ArrayList is given to pass two to create the output files.

In Pass two, the class parses through each line of assembly, which is passed in through the ArrayList, and creates the appropriate record for the output file. In addition to creating the output file, the program creates a listing file as well that gives the user the value loaded into memory. Helper methods are called based on the instruction defined in the operational field and are parsed into hex strings by the elements defined in the operands field. On top of creating the output files, the class checks for errors to ensure constants and symbols are correct values within the appropriate ranges.

The diagram below gives a quick glimpse of how the input file feeds into the different classes and gives the desired output file.


Assembly input file
Input
App.java
Validator.java
Creates an (separating each line of assembly) ArrayList<List<String>
Input
Pass1.java
Creates SymbolTable and LiteralTable (HashMaps)
Pass2.java
Input
Creates object file and listing file

The UML diagram below shows the interaction between the different classes, with their attributes (global members) along with their operations (methods).



Individual Components

Validator

```
Validate(MOT.Machine_Op_Table mot) {
```

The constructor to the validator takes in an initialized machine operation table and assigns it to an instance variable, it then uses the instance variable MOT to validate operations.

```
List<String> validate(String line, ArrayList<List<String>> lines) throws Exception {
```

The validate method is the main method to interface with the validator class. It takes in a “line” and the target “lines” ArrayList, it validates the line and ensures the whole program doesn’t have duplicate symbols using “lines”.

```
String stripComment(String line) {
```

This method simply takes any comments out of the String “line” and returns an updated String without comments.

```
boolean containsAnyKey(String input) {
```

The containsAnyKey method checks to see if the String “input” contains an operation that is in the MOT.

```
void validateComment(String line) throws Exceptions.CommentLineNoSemicolon {
```

The validateComment method ensures that a comment line starts with a semicolon

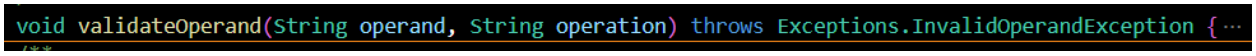
```
void validateOperation(String operation, OpType type) throws Exception {
```

The validateOperation method ensures the operation is either a pseudo operation or a machine operation.

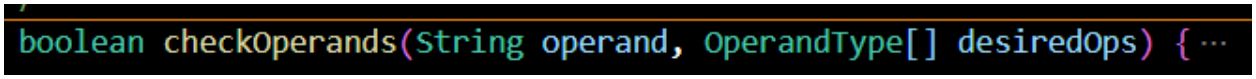
```
void validateInstruction(String line) throws InvalidOperandException {
```

The validateInstruction method takes in a line and confirms it’s syntactically valid and has the correct number of operands.

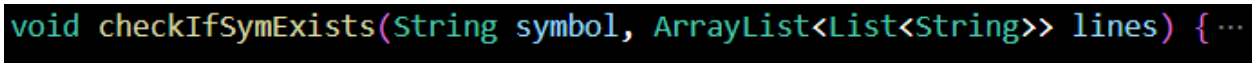
The stripArray method returns a List of everything in a line broken into its component parts (label, operation, operands), with comments taken out.



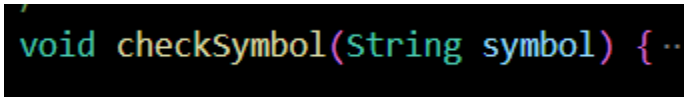
The validateOperand method Determines if the operation is a Machine OP and then verifies its operand length and order is as expected according to the machine's description.



The checkOperands method takes in an operand and what type of input it takes, for example the operand could be “AND” and the OperandType could be {Register, Register, Register}, and the method would return “true” if the operands syntactically match the operand types and “false” otherwise.

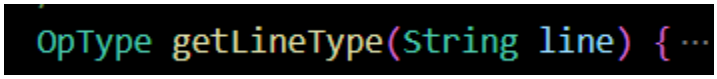


The checkIfSymExists method checks to see if a symbol already exists in the processed lines.

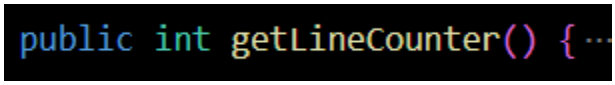


The checkSymbol method ensures a symbol is syntactically valid.

The validateAndSplitLine method splits a line and verifies it has all five parts based in the following order: Label, White Space, Operation, White Space, Operand/Comment.



The getLineType method returns an ENUM (OpType) describing the type of line passed



The getLineCounter method returns the current line count.

```
void validateEntOrExt(String line, List<String> list) throws CommentLineNoSemicolon {
```

The validateEntOrExt will validate if an ENT or EXT line has proper syntax. It takes in the line to validate and a list which contains a line split with the following order “Symbol, Operation, Operand/Comment”

Pass One

```
private static ArrayList<String> LitArray = new ArrayList<String>();
```

A private static ArrayList meant to hold the literals while the symbol table is being filled

```
public static int fillTables(ArrayList<List<String>> arr, int loc, HashMap<String, String[]> SymTable,
    HashMap<String, Integer> LitTable) {
```

Initial method to fill the symbol and literal tables

```
public static int fillSymTable(ArrayList<List<String>> arr, HashMap<String, String[]> SymTable, int loc) {
```

Loops through the ArrayList to create the symbol table. It checks if there is a symbol in the line, and if there is, calls a method to fill the array of Strings to be the value of the HashMap before putting the symbol with the array into the symbol table HashMap. Also checks if the operand of the line contains a literal value, and if it does, puts the literal into an ArrayList to be used after the symbol table is filled. Finally, increments the location counter each iteration, and after the full loop is done, returns the fully updated location counter.

```
public static String[] fillSymArr(String sym, String instruct, String operand, HashMap<String, String[]> SymTable,
```

Creates a blank, two-value array of Strings to be filled and returned. The method then checks the instruction to see if it is a .EQU instruction or some other instruction. If it is not a .EQU instruction, it fills the first element of the symbol array with the current location counter and sets the second element to “R” for relative. If it is a .EQU instruction, it fills the symbol array by calling another method that deals with absolute values.

```
public static int fillLitTable(HashMap<String, Integer> LitTable, int locCount, ArrayList<String> litArray) {
```

This method is called after the symbol table is fully filled. Taking the empty literal table, the current location counter, and the stored array of literals to be placed into the literal table, this method loops through the array of stored literals, placing them into the literal table with the current location counter as the value in the HashMap. The location counter is incremented every time by one, as each literal only takes up a single space in memory.

```
public static String[] getEQUVal(String sym, String operand, HashMap<String, String[]> SymTable) {
```

Creates a blank, two-value array of Strings to be filled and returned. The method checks if the value is a hex or decimal value, and if it is, sets the first element of the array to the hex or decimal value provided, and the second value to “A” for absolute. If it is not a hex or decimal value, then it means that the operand is a symbol, and it then checks if that symbol is absolute or relative, sets that value of the new symbol array to the same, and takes the value of that symbol and places it as the value of the new symbol. Finally, it returns the symbol array to be used by the FillSymArr() method.

```
public static int incLoc(String instruct, String operand, int locCount) {
```

Takes in the current location counter, the instruction, and the operand in order to properly increment the location counter. If the instruction is a .STRZ, the location counter is incremented by one more than the length of the string in the operand. If the instruction is a .BLKW, it is incremented by the value of the operand. Otherwise, if the instruction is not a .EQU, the instruction counter is incremented by one. The reason why .ORIG and .END are left out in that, along with .EQU, the location counter is not incremented at all by these three pseudo-ops. However, with .ORIG and .END, since it is known that these will always be the first and last instructions of the program in a valid file, we are able to start the initial loop through the ArrayList one line in and end one line before the end to preclude these two instructions.

Pass Two

Purpose of the component

This class serves as the final checkpoint for converting the assembly file into an object file, accompanied by a listing file that contains the conversion between the assembly and object file. As per the object file to be created, the class deals with parsing each line of assembly, creating a header record containing the segment name, size, and start address, text records for the machine instructions, and the end record to denote the start of the program execution. Moreover, the class also creates the listing file which outputs in the following format:

```
(Addr hex) | Contents | Contents | (line #) | Label | Instruction | Operands
           |   hex    |  binary  |          |       |             |
```

Here, the “contents” are the data loaded for a given address of memory. The “line” refers to the line of assembly code and the “label, instruction, operands” together form a single line of assembly code.

General logic structure

All the information required by Pass Two is passed to the constructor of the method. This includes the stream for writing the file, the different tables, and the segment size. From here, a method is called to parse the input where the input file is sent over in the form of a List<List<String>>. The components of the list are split as the following:

Index	Component
0	Label (if any)
1	Operational field (Opcode)
2	Operand (if any)
3	Line number (with relation to the assembly)

The App.java class ensures the inputs are present or puts a blank space in these areas if absent. The given input list would not contain any whitespaces or comments. The parse function iterates through the entire list, parses each line of assembly, validates for any errors (mentioned more in the Implementation section of Pass Two), creates the desired record file, and writes those contents to the object and listing file.

When parsing through each instruction, a look-up table is used to determine the type of instruction to parse (machine vs pseudo-op) and after finding the desired value, helper methods are called to get the text record for that line of assembly code. As the record is created, any potential errors with the line of code are checked to ensure the wrong thing is not encountered. If so, then an error is thrown to the user saying which line of assembly is faulty and termination of further execution.

If the text record was successfully created, it is sent to another method to output to an object file and listing file. If the program successfully wrote to the stream, it continues with the next line of assembly. After all the lines of assembly are parsed, the output streams are closed for the written records to appear.

Implementation

Pass Two holds a few private members that are initialized after an object is instantiated in the main class. Some of the private members include a location counter to track the address at which a record is to be stored, a Machine-Ops table for parsing any machine instructions, a Pseudo-Ops table for parsing any assembler instructions, a stream for the object file, a stream for the listing file, an instance of the String_Parser class (more details about this class mentioned in the String

Parser section), the segment size, the symbol table, the literal table, a flag to check when a program is relocatable, and the initial load address.

For Pass Two, the input required for the class includes the Machine-Ops table, the Pseudo-Ops table, the symbol table, the literal table, streams for writing the object and listing file, and the size of the segment. The actual representation of these components can be seen below:

```
public Pass2(Machine_Op_Table mot, Pseudo_Op_Table pot, Map<String, String[]> symbolTable,
        Map<String, Integer> literalTable, FileWriter objectFile, FileWriter listingFile, int segmentSize) {
```

Each of the private members mentioned above is initialized in the constructor. Most of the parameter values are stored as a reference in the class to be accessed globally.

The next main structure to pass 2 is the method parseMethod(). The method header is given below.

```
public void parseInput(List<List<String>> list) throws Pass2Exception {
```

The parse method takes in an input the lines of assembly split into labels, opcodes, operands, and the line number. The method first iterates through each element of the list gets these components and checks what to call next. If the given opcode field is a machine instruction, then the method for assembling the machine ops is called. The method signature is given below.

```
public String assembleMachineOpLine(String opcode, String operand, String line) throws Pass2Exception
```

The method accepts the instruction name, the operand field, and the line number. The operand field is split based on the commas (each operand field turns into its value in the array). Once split, the enumerated name of the machine instruction is obtained to cycle through all the possible values present for a machine instruction. Each of the methods returns the body of the text record. The method concatenates a “T” at the start to account for the text record and a new line at the end for the next text record to be placed. An example of the first few instructions being parsed is given below.

```
switch (ins_name) {
case ADD:
    record += addOp(splitOperand, value, size, line) + "\n";
    break;
case AND:
    record += andOp(splitOperand, value, size, line) + "\n";
    break;
case BR:
    record += brxOp(splitOperand, "0", "0", "0", value, size, line) + "\n";
    break;
case BRN:
    record += brxOp(splitOperand, "1", "0", "0", value, size, line) + "\n";
    break;
```

Using a switch-case block for each of the machine instructions present would improve the efficiency of the code. Using an if-else block could end up being O(n) time when parsing through every line. A switch case gets hashed in compile time and should be close to O(1). Each case in the switch-case calls upon the respective helper method to create the text record, which is geared to follow the syntax for said instruction. Dynamically accounting for the parsing of the instruction could be more complicated and possibly inefficient than "brute force" as the factors that distinguish the instructions are sparse and accounting for a variable length array (operands without the comments or junk values) can vary and need to be accounted for.

If the parseInput() method failed to identify a machine instruction, then the opcode would be a Psueod Operation (otherwise the Validator.java class would throw an error). The method signature for assembling the Pseudo-Ops is given below.

```
public String assemblePseudoOpLine(String symbol, String opcode, String operand, String line)
        throws Pass2Exception {
```

The use for switch-case on the Pseudo-Ops is similar to the one mentioned for the Machine-Ops, but the text record formation is performed in the specified helper method as some of the instructions don't occupy space in memory, while others occupy more than one block of memory.

```
switch (ins_name) {
case BLKW:
    record += blkwOp(symbol, opcode, operand, line);
    break;
case END:
    record += endOp(operand, line);
    break;
```

When trying to parse a line of assembly that involves register usage without addressing, the pseudocode below gives a summary of the typical structure followed for each instruction.

1. Get the address from LC, convert it to a hex string (ensure the hex is 4 strings long), add it into the hex record, and increment LC by the size defined
2. Convert the opcode_value to a binary string (ensure the binary is 4 strings long), and add to binary string
3. Iterate through the split operand times length
    a. Is the first character a register?
        i. If it is iteration three, add three 0s
        ii. Convert the register value to an integer (ensure the value is between 0-7)
        iii. Convert the register value to a binary string (ensure the binary string is 3 strings long)
        iv. Add the value to the binary string
    b. Is the first character a hex or decimal? (only possible at the end)
        i. Add a 1 to the binary string
        ii. Parse the immediate value (ensure it is within immediate range)
        iii. Convert to a binary string and add it to the binary string (ensure the binary string is 5 strings long)
    c. Otherwise, it is a symbol
        i. Ensure the symbol table contain the symbol and it is an absolute symbol
        ii. Is it the third iteration?
            1. Convert the symbol value to an integer (ensure the value is within immediate range)
            2. Convert the symbol value to a binary string (ensure the binary string is 5 strings long)
        iii. Otherwise:
            1. Convert the symbol value to an integer (ensure the value is between 0-7)
            2. Convert the symbol value to a binary string (ensure the binary string is 3 strings long)
            3. Add the value to the binary string
4. Convert the binary string to an unsigned integer
5. Convert the integer to a hex string (ensure the hex is 4 strings long)
6. Add the value to the text record

When trying to parse a line of assembly that involves addressing, the pseudocode below gives a summary of the typical structure followed for each instruction.

1. Get the address from LC, convert it to a hex string (ensure the hex is 4 strings long), add it into the hex record, and increment LC by the size defined

2. Convert the opcode_value to a binary string (ensure the binary is 4 strings long), and add to binary string
3. Set the first three bits with the parameter values/don't care/registers
4. Set the page offset for binary string
    a. Check if the last value is a symbol
        i. Verify if the symbol is in the symbol table
        ii. Parse the value for the given symbol
        iii. Determine if the symbol is relative or absolute
    b. Try to parse the decimal or hex value and verify it is within address range
    c. Check whether the value formed from either the symbol or hex/decimal is on the same page as the PC (page(this.LC + 1) == page(address))
    d. Store the lower 9 bits of the address for the binary string
5. Convert the binary string to an unsigned integer
6. Convert the integer to a hex string (ensure the hex is 4 strings long)
7. Add the value to the text record
8. If the program is relocatable and the address formed was through a relative symbol, add a modification record (denoting the pgoffset 9 changes)

Although the general structure is very similar, some helper methods replicate the same method with a few modifications for the sake of easier understandability. The list of instructions given below calls the same method that contains the same logic without any changes compared to the instructions that are its own method.

Separate methods	Dependent methods
ADD	AND
BRx	-----
DBUG	-----
JSR	JMP
JSRR	JMPR
LD	LDI, LEA, ST, STI
LDR	STR
NOT	-----
RET	-----
TRAP	-----

Each of the Pseudo-Ops are all separate methods as each operation has a separate requirement to fulfill.

The first part of creating the header record is obtaining the address and placing it in the record. When calling the helper method, the method parses the address into a hex string, increments the location counter by the size defined by the instruction, and the string address is sent to the caller. The next line calls a String Parser method which converts the current opcode into a binary string to be added to the text record.

```
String lcAddress = addressFromLC(size);
textRecord += lcAddress;

String binaryString = sp.opcodeBinaryString(opcode_value);
```

One of the few possible operands to be found in this field is registers. When a register is found, the value after the 'R' is taken and parsed into a string (if possible) and added to the binary string. The register value is treated as a decimal value to ensure the parsing works (the parsing is designed specifically for the abstract machine to deal with decimals and hex) In the parsing method, it is checked whether the register value is within the specified range, otherwise a Pass 2 error is thrown to the user with a message saying the value is not within range.

```
if (pos.charAt(0) == 'R') {
    // Convert the register value to fit the binary
    String regVal = sp.registerToBinaryString("#" + pos.substring(1), line);
    binaryString += regVal;
}
```

Another possible operand would be the direct usage of constants for fields like immediate, index, etc. Similar to registers, a parsing method is called to ensure the value of the constant is within the required range, otherwise, an error is thrown. The occurrences of constants would occur in immediates, indexes, trap vectors, and addresses.

```
if (pos.charAt(0) == '#' || pos.charAt(0) == 'x') {
    String trap = sp.trapToBinaryString(pos, line);
    binaryString += trap;
}
```

Absolute symbols are potential values that replace any of the fields (i.e, registers, traps, etc. can be replaced with absolute symbols). The method header for whether the element of the operand (pos) is a valid symbol to use.

```
private void absoluteSymbolTable(String pos, String line) throws Pass2Exception
```

When an absolute symbol is placed to replace a certain element of the operand, the symbol needs to check whether the symbol exists in the symbol table and whether the symbol is an absolute symbol. An error is thrown for each respective field not satisfied. Once the symbol is verified, it is parsed into the required field which returns the binary version of the integer value if the symbol's value is within the specified range.

```
absoluteSymbolTable(pos, line);

// Convert the register value to fit the binary
String regVal = sp.registerToBinaryString(symbolTable.get(pos)[0], line);
binaryString += regVal;
```

With any address operations, the operand is mainly checked for the type of symbol to use, if the value given for this element is within the proper range, and if the address is in the same page range as the location counter (as the address formed for these operations are taking the upper bits of the program counter with the lower 9 bits provided in the binary string (page offset)). When using symbols for an address field of an operand, relative symbols are values that are allowed. The method header for checking if a symbol is relative is given below.

```
private boolean isRelativeSymbol(String pos, String line) throws Pass2Exception {
```

An error is thrown to the user only when the symbol is not found in the symbol table. After getting the value of the address, the page at which the address is defined is checked against the location counter as the conversion to binary is the lower nine bits of the given address value. An error is thrown to the terminal if the page range doesn't match or the address value is invalid.

```
String offset = sp.pageRangeAndOffset(lcAddress, address, line);
binaryString += offset;
```

For an address field, if an address field is replaced with a relative symbol, the text record can be relocatable. A program becomes relocatable if an address was not defined for the **.**ORIG operand portion.

```
if (this.isRelocatable && relative) {
    textRecord += "M9";
}
```

The end of each text record is converting the sixteen-bit binary string back to a hex string and the value is added to the text record and sent to the caller.

```
String hexString = sp.convertBinaryToHexString(binaryString, line);
textRecord += hexString;
```

The text records for Pseudo-Ops are similar to parsing addresses as the operand only accepts one operand. Most of the checking is making sure the provided value is within the specific range, otherwise an error is thrown to the user. After the text records are parsed, they are sent to the output stream for the object file and called the method for making the listing file output. The method header for the listing file output is given below.

```
public String outputToListingFile(String symbol, String opcode, String operand, String assemblyLine, String line)
```

However, the listing file output differs for three different cases. These cases are for the **.**STRZ instruction, **.**BLKW instruction, and literals. In these cases, the string operation requires the output of several text records, the block requires a set of memory allocated, and the literals are added to the end of the file. Each of the method's headers is given below.

```
public void blkwListingFile(String symbol, String address, String opcode, String operand, String line)
        throws Pass2Exception {
```

```
public void strzListingFile(String symbol, String opcode, String operand, int i, String assemblyLine, String line)
        throws Pass2Exception {
```

```
public String literalListingFile(String literal, Integer value, String line) throws Pass2Exception
```

Before the end of the segment is reached, literals have to be dealt with, so a method is called to iterate through every component in the literal table and added to the object file before the end record is printed.

```
public String literalObjectFile(String literal, Integer value, String line) throws Pass2Exception {
```

Once all the components are created, the output streams are closed for the actual data to get displayed.

String Parser

Purpose of the component

The String Parser allows for Pass Two to parse between strings and integer values of any constants, address, etc. for the abstract machine. Dealing with numerical parsing separate from record parsing would abstract how each component work and wouldn't need to deal with the programming of numerical parsing (lower abstraction than parsing records).

Implementation

The implementation for parsing a given string requires that the given value is either a hex, decimal, or literal. The method header given below allows the user to see whether an invalid value can be parsed and thrown appropriate messages when the value is the expected result.

```
public boolean canParseInt(String input)
```

The actual implementation of canParseInt() relies on the helped method which uses the Integer class isInteger() method to check whether the input can be parsed surrounded by a try-catch block to prevent number format exceptions. If the parse was successful, it ensures the value returned to the user is within the possible sixteen-bit ranges for the abstract machine.

```
private boolean isInteger(String input, int radix) {
```

There are several methods in the class that allows for parsing a constant value, catered specifically for certain types such as immediates, indexes, etc. After the value was parsed, the possible ranges for which the value can be is checked and if it matches the necessary ranges, it returns the parsed values. Otherwise, it would return an error value.

```
public int parseImmediate(String input)
```

Another common parsing found throughout the process of making the object file is converting between integer values to a binary string. Each of the hex instructions is stored in four characters, the binary string is easier to build with as it would contain sixteen bits in it. Once the binary string was derived, the value is converted to a hex string to satisfy the output file format. All the conversions are done using the Integer class conversions to a certain string and accounted for any error values and appending zeroes to the front to be a four characters hex.

```
public String indexToBinaryString(String ind, String line) throws Pass2Exception {
```

```
public String convertBinaryToHexString(String binString, String line) {
```

For parsing literal values, the literal table is passed in to ensure the literal exists and when parsing, the value after the “=” is extracted and the integer is parsed to ensure it is within the range and value associated with that literal (address) is sent to the caller.

```
public String literalValue(Map<String, Integer> literalTable, String address, String line) throws Pass2Exception {
```

Modified Appendix

Data Structures

The modified assembler keeps most of the data structures similar, except for the Pseudo-Ops Table (POT). The POT has additional information stored to account for the new external symbols assembler ops.

Additions to the Pseudo Operation Table (POT)

The POT's method, `loadTable()`, loads the possible instructions along with their format followed, the size of the opcode, and the enumerated name of the instruction. To add to the current list, the "**.**ENT" and "**.**EXT" instructions are added.

```
// Linking/Loader ops
this.table.put(".ENT", new Pseudo_Op_Info(Pseudo_Op_Ins_Enum.ENT, ZERO, Pseudo_Op_Format.DEFINITE));
this.table.put(".EXT", new Pseudo_Op_Info(Pseudo_Op_Ins_Enum.EXT, ZERO, Pseudo_Op_Format.DEFINITE));
```

These instructions have a definite length in the assembler, with a size of zero. Here, the size denotes how much space each instruction takes in memory.

Individual Components

The general algorithm followed by each component is the same, with a few modifications. The Validator checks if the format for the external symbols op is followed correctly and Pass 2 creates new modification records for the entry and external symbols.

Validator

In order to accept .ENT and .EXT we must add it as a OpType that can occur

```
enum OpType {
    COMMENT, ORIG, END, EQU, FILL, STRZ, BLKW, INSTRUCTION, UNKNOWN, ENT, EXT
}
```

Modifications to `getLineType(String line)` will be as follows

```
} else if (line.contains(OpType.ENT.toString())) {
    return OpType.ENT;
} else if (line.contains(OpType.EXT.toString())) {
    return OpType.EXT;
} else {
```

We've create a method to verify the syntax of a line that with ENT or EXT pseudo op

```
void validateEntOrExt(String line, List<String> list) throws CommentLineNoSemicolon {
    line = line.trim();


    if(!list.get(0).isBlank()) {
        throw new IllegalArgumentException("No Label Allowed for ENT or EXT operations");
    }

    if(list.get(1).trim().charAt(0) != ';') {

        String[] symbols = list.get(2).split(",");

        for(String s: symbols) {
            checkSymbol(s.trim());
        }

    }
}
```

This method's arguments are the entire pseudo op line and a list of strings of size 3 with the label, operation, and operand fields in that order

Additionally, we've added a global variable to keep track of lines that are comments and ENT/EXT pseudo ops.

```
private int commentOrEntExtCounter = 1;
```

Finally, inside the `validate()` method we've made the following change

```
if (type == OpType.ENT || type == OpType.EXT) {
    // validate line content
    // make sure line above it is .ORIG or .EXT or .ENT
    if(lineCount - this.commentOrEntExtCounter != 2) {
        throw new IllegalArgumentException("ENT and EXT must be declared right after ORIG");
    }
    validateEntOrExt(line,result);
    this.commentOrEntExtCounter++;
```

This condition will check if an ENT or EXT line is placed right after the ORIG pseudo op. If so it will call the `validateEntOrExt()` method to validate the line.

Pass 2

As the assembler is allowed to use symbols from different files, any of these symbols are defined in the operand field of the **.**EXT op is stored in a temporary external symbol table.

```
private Map<String, String> tempExternalTable;
```

Modifications to the `assemblyPseudoOpLine()` method are two additional helper methods for parsing the entry and external pseudo-ops.

```
public String assemblePseudoOpLine(String symbol, String opcode, String operand, String line)
            throws Pass2Exception {
```

```
case ENT:
        record += entOp(splitOperand, line);
        break;
case EXT:
        record += extOp(splitOperand, line);
        break;
```

Each of the methods allows for parsing the operand fields of the entry and external ops for creating the necessary N-records and the external symbols used. Each of the operands is split by the number of elements contained, stored in an array, and sent to the method. The methods iterate over the length of the array and do the following:

1. For **.**ENT op
    a. Append an “N” to the record
    b. Check if the given symbol is a relative symbol
    c. Get the value of the symbol from the symbol table
    d. Append the symbol with its hex value to the record

2. For **.**EXT op
    a. Get the symbol from the operand
    b. Check if the symbol is not defined in the symbol table
    c. Add the symbol to the temporary external symbol table

```
private String entOp(String[] operand, String line) throws Pass2Exception
```

```
private String extOp(String[] operand, String line) throws Pass2Exception
```

The helper methods used for checking if the symbol is absolute or relative contain additional checks to account for the external symbols. For the `absoluteSymbolTable()` method, the method also checks if an external symbol is not used in the operand fields (external symbols are relative) that only accept absolute symbols. For the `isRelativeSymbol()` method, the

method signature changed to an integer to account for whether a relative symbol from the program segment is used or an external symbol is used.

```
private void absoluteSymbolTable(String pos, String line) throws Pass2Exception
```

```
private int isRelativeSymbol(String pos, String line) throws Pass2Exception
```

```
if (this.tempExternalTable.containsKey(pos)) {
        return 2;
} else if (symbolTable.get(pos)[1].equals("R")) {
        return 1;
}
```

Once determined if the symbol used is relative and relocatable, the appropriate modification record is appended to the end of the text record. As mentioned previously, only the instructions containing an address field in their operand will contain modification records. This includes BRx, JSR, JMP, LD, LDI, LEA, ST, STI, and **.**FILL ops.

For machine instructions that include address fields in their operands that are relocatable, X-records with nine-bit modifiers are appended to the end. If the symbol is external, then the name of the symbol is appended to the end of the record. If the symbol is a symbol found in the symbol table, then the name of the segment is appended to the record.

```
if (this.isRelocatable) {
        if (relative == 1) {
                textRecord += ("X9" + this.segName);
        } else if (relative == 2) {
                textRecord += ("X9" + operand[0]);
        }
}
```

The **.**FILL op follows the same logic as machine instructions, except for the bit-offset of the modification record. Since the **.**FILL op only requires a single field, all sixteen bits of the record can change if it is relocatable.

```
if (this.isRelocatable) {
        if (relocatable == 1) {
                textRecord += ("X16" + this.segName);
        } else if (relocatable == 2) {
                textRecord += ("X16" + operand);
        }
}
```

The assembler accounts for forward referencing through external symbols. If a symbol's value is an external symbol, the address is replaced with the value in the external symbol table and is appended with an X-record with the external symbol's name.

Code Appendices

Machine-Op Table Package

Machine_Op_Info.java

```
package lab3_integrated.assembler.MOT;

public class Machine_Op_Info {
    /*
     * Enumeration to store the mnemonic name of the instruction.
     */
    private Machine_Op_Ins_Enum ins;

    /*
     * Value for the given opcode.
     */
    private int opcode;

    /*
     * Length of the instruction.
     */
    private int size;

    /**
     * Default constructor to store the opcode value, word size, and
specific format
     * for a given machine instruction.
     *
     * @param ins    the enumerated name of the instruction
     * @param opcode the opcode's decimal value
     * @param size   the word size of the instruction
     * @param format the format of the given instruction
     */
    public Machine_Op_Info(Machine_Op_Ins_Enum ins, int opcode, int size)
{
        this.ins = ins;
        this.opcode = opcode;
        this.size = size;
    }
```

```
    /**
     * Returns the enumerated version of the mnemonic op name.
     *
     * @return mnemonic name
     * @ensures getInstructionName = this.ins
     */
    public Machine_Op_Ins_Enum getInstructionName() {
        return this.ins;
    }

    /**
     * Returns the decimal opcode value.
     *
     * @return opcode value
     * @ensures getOpcode = this.opcode
     */
    public int getOpcode() {
        return this.opcode;
    }

    /**
     * Returns the decimal size value.
     *
     * @return size value
     * @ensures getSize = this.size
     */
    public int getSize() {
        return this.size;
    }
}
```

Machine_Op_Ins_Enum.java

```
package lab3_integrated.assembler.MOT;

// Enumeration for the names of each instruction
```

```
public enum Machine_Op_Ins_Enum {
    ADD, AND, BR, BRN, BRZ, BRP, BRNZ, BRNP, BRZP, BRNZP, DBUG, JSR, JMP,
JSRR, JMPR, LD, LDI, LDR, LEA, NOT, RET, ST,
    STI, STR, TRAP
}
```

Machine_Op_Table.java

```
package lab3_integrated.assembler.MOT;

import java.util.HashMap;

public class Machine_Op_Table {
    /**
     * Constants for the opcode values of each instructions.
     */
    public static final int BRX_OPCODE = 0; // Opcode value for the BRx
instruction
    public static final int ADD_OPCODE = 1; // Opcode value for the ADD
instruction
    public static final int LD_OPCODE = 2; // Opcode value for the LD
instruction
    public static final int ST_OPCODE = 3; // Opcode value for the ST
instruction
    public static final int JSR_OPCODE = 4; // Opcode value for the JSR
instruction
    public static final int AND_OPCODE = 5; // Opcode value for the AND
instruction
    public static final int LDR_OPCODE = 6; // Opcode value for the LDR
instruction
    public static final int STR_OPCODE = 7; // Opcode value for the STR
instruction
    public static final int DBUG_OPCODE = 8; // Opcode value for the DBUG
instruction
    public static final int NOT_OPCODE = 9; // Opcode value for the NOT
instruction
    public static final int LDI_OPCODE = 10; // Opcode value for the LDI
instruction
```

```
    public static final int STI_OPCODE = 11; // Opcode value for the STI
instruction
    public static final int JSRR_OPCODE = 12; // Opcode value for the JSRR
instruction
    public static final int RET_OPCODE = 13; // Opcode value for the RET
instruction
    public static final int LEA_OPCODE = 14; // Opcode value for the LEA
instruction
    public static final int TRAP_OPCODE = 15; // Opcode value for the TRAP
instruction

    /*
     * Constants for the instruction size.
     */
    public static final int ONE = 1; // Size of each opcode
    public static final int INVALID = -1; // Invalid values for when a
requirement is not met

    /**
     * Map to represent the machine op table.
     */
    private HashMap<String, Machine_Op_Info> table;

    /**
     * Default constructor.
     */
    public Machine_Op_Table() {
        this.table = new HashMap<>();
        loadTable();
    }

    /**
     * Loads the 16 machine instructions into the Machine-Ops Table with
their size,
     * opcode value, and format they are stored as (page offset, index, or
none).
     *
     * @ensures Machine_Op_Table = <"opcode name", <value, size, format>>,
for all
     *          instructions
```

```
     */
    public void loadTable() {
        // Data processing instructions
        this.table.put("ADD", new Machine_Op_Info(Machine_Op_Ins_Enum.ADD,
ADD_OPCODE, ONE));
        this.table.put("AND", new Machine_Op_Info(Machine_Op_Ins_Enum.AND,
AND_OPCODE, ONE));
        this.table.put("NOT", new Machine_Op_Info(Machine_Op_Ins_Enum.NOT,
NOT_OPCODE, ONE));

        // Data movement instructions: Load instructions
        this.table.put("LD", new Machine_Op_Info(Machine_Op_Ins_Enum.LD,
LD_OPCODE, ONE));
        this.table.put("LDI",
                new Machine_Op_Info(Machine_Op_Ins_Enum.LDI, LDI_OPCODE,
ONE));
        this.table.put("LDR", new Machine_Op_Info(Machine_Op_Ins_Enum.LDR,
LDR_OPCODE, ONE));
        this.table.put("LEA",
                new Machine_Op_Info(Machine_Op_Ins_Enum.LEA, LEA_OPCODE,
ONE));

        // Data movement instructions: Store instructions
        this.table.put("ST", new Machine_Op_Info(Machine_Op_Ins_Enum.ST,
ST_OPCODE, ONE));
        this.table.put("STI",
                new Machine_Op_Info(Machine_Op_Ins_Enum.STI, STI_OPCODE,
ONE));
        this.table.put("STR", new Machine_Op_Info(Machine_Op_Ins_Enum.STR,
STR_OPCODE, ONE));

        // Flow of control instructions: Branching (all valid
combinations)
        this.table.put("BR", new Machine_Op_Info(Machine_Op_Ins_Enum.BR,
BRX_OPCODE, ONE));
        this.table.put("BRN",
                new Machine_Op_Info(Machine_Op_Ins_Enum.BRN, BRX_OPCODE,
ONE));
        this.table.put("BRZ",
```

```
                new Machine_Op_Info(Machine_Op_Ins_Enum.BRZ, BRX_OPCODE,
ONE));
        this.table.put("BRP",
                new Machine_Op_Info(Machine_Op_Ins_Enum.BRP, BRX_OPCODE,
ONE));
        this.table.put("BRNZ",
                new Machine_Op_Info(Machine_Op_Ins_Enum.BRNZ, BRX_OPCODE,
ONE));
        this.table.put("BRNP",
                new Machine_Op_Info(Machine_Op_Ins_Enum.BRNP, BRX_OPCODE,
ONE));
        this.table.put("BRZP",
                new Machine_Op_Info(Machine_Op_Ins_Enum.BRZP, BRX_OPCODE,
ONE));
        this.table.put("BRNZP",
                new Machine_Op_Info(Machine_Op_Ins_Enum.BRNZP, BRX_OPCODE,
ONE));

        // Flow of control instructions: Jumping
        this.table.put("JSR",
                new Machine_Op_Info(Machine_Op_Ins_Enum.JSR, JSR_OPCODE,
ONE));
        this.table.put("JMP",
                new Machine_Op_Info(Machine_Op_Ins_Enum.JMP, JSR_OPCODE,
ONE));
        this.table.put("JSRR",
                new Machine_Op_Info(Machine_Op_Ins_Enum.JSRR, JSRR_OPCODE,
ONE));
        this.table.put("JMPR",
                new Machine_Op_Info(Machine_Op_Ins_Enum.JMPR, JSRR_OPCODE,
ONE));

        // Flow of control instructions: General
        this.table.put("RET", new Machine_Op_Info(Machine_Op_Ins_Enum.RET,
RET_OPCODE, ONE));
        this.table.put("TRAP", new
Machine_Op_Info(Machine_Op_Ins_Enum.TRAP, TRAP_OPCODE, ONE));

        // Miscellaneous instructions
```

```
        this.table.put("DBUG", new
Machine_Op_Info(Machine_Op_Ins_Enum.DBUG, DBUG_OPCODE, ONE));
    }

    /**
     * Returns the word size for the instruction, or -1 if the instruction
is
     * invalid.
     *
     * @param key Mnemonic name for the instruction
     * @return word size of the key instruction or -1 if the instruction
is invalid
     * @ensures getSize = word_size(key) or -1
     */
    public int getSize(String key) {
        if (!this.table.containsKey(key)) {
            return INVALID;
        }

        Machine_Op_Info op = this.table.get(key);
        return op.getSize();
    }

    /**
     * Returns if the given instruction is in the Machine-Ops Table
     *
     * @param key Mnemonic name for the instruction
     * @return the boolean value for if the given instruction is in the
Machine-Ops
     *         Table or not
     * @ensures containsOp = Machine_Op_Table contains key (true), or
false
     */
    public boolean containsOp(String key) {
        return this.table.containsKey(key);
    }

    /**
     * Returns the value for the instruction, or -1 if the instruction is
invalid.
```

```
     *
     * @param key Mnemonic name for the instruction
     * @return opcode value of the key instruction or -1 if the
instruction is
     *         invalid
     * @ensures getOpcode = opcode_value(key) or -1
     */
    public int getOpcode(String key) {
        if (!this.table.containsKey(key)) {
            return INVALID;
        }

        Machine_Op_Info op = this.table.get(key);
        return op.getOpcode();
    }

    /**
     * Returns the enumerated version of the instruction name, or null if
it is
     * invalid.
     *
     * @param key Mnemonic name for the instruction
     * @return enumerated version of the key or null for invalid
instructions
     * @ensures getInstructionName = enum_name(key) or null
     */
    public Machine_Op_Ins_Enum getInstructionName(String key) {
        if (!this.table.containsKey(key)) {
            return null;
        }

        Machine_Op_Info ins = this.table.get(key);
        return ins.getInstructionName();
    }
}
```

Pseudo-Op Table Package

Pseudo_Op_Format.java

```
package lab3_integrated.assembler.POT;

// Enumeration for the format of each instruction
public enum Pseudo_Op_Format {
    DEFINITE, VARIABLE;
}
```

Pseudo_Op_Info.java

```
package lab3_integrated.assembler.POT;

public class Pseudo_Op_Info {
    /*
     * Enumeration to store the mnemonic name of the Pseudo-op.
     */
    private Pseudo_Op_Ins_Enum ins;

    /*
     * Length of the instruction.
     */
    private int length;

    /*
     * Enumeration to store the type of format (page offset, index offset,
or none).
     */
    private Pseudo_Op_Format format;

    /**
     * Default constructor to store the length and the format for a given
Pseudo-op.
     *
     * @param ins    enumerated name of the pseudo-op
```

```
     * @param length definite/variable size for the pseudo-op
     * @param format format in the type of format to define the length of
the
     *               pseudo-op
     */
    public Pseudo_Op_Info(Pseudo_Op_Ins_Enum ins, int length,
Pseudo_Op_Format format) {
        this.ins = ins;
        this.length = length;
        this.format = format;
    }

    /**
     * Returns the enumerated version of the mnemonic op name.
     *
     * @return mnemonic name
     * @ensures getInstructionName = this.ins
     */
    public Pseudo_Op_Ins_Enum getInstructionName() {
        return this.ins;
    }

    /**
     * Returns the decimal size value.
     *
     * @return size value
     * @ensures getSize = this.length
     */
    public int getLength() {
        return this.length;
    }

    /**
     * Returns the format specified under the enumeration.
     *
     * @return format value
     * @ensures getFormat = this.format
     */
    public Pseudo_Op_Format getFormat() {
        return this.format;
```

```
    }
}
```

Pseudo_Op_Ins_Enum.java

```
package lab3_integrated.assembler.POT;

// Enumeration for the names of each instruction
public enum Pseudo_Op_Ins_Enum {
    ORIG, END, EQU, FILL, STRZ, BLKW, ENT, EXT
}
```

Pseudo_Op_Table.java

```
package lab3_integrated.assembler.POT;

import java.util.HashMap;

import lab3_integrated.assembler.Passes.String_Parser;

public class Pseudo_Op_Table {
    /*
     * Constants for the instruction size.
     */
    public static final int ZERO = 0; // Pseudo-Ops with size 0/variable
length (initial value)
    public static final int ONE = 1; // Pseudo-Ops with size 1
    public static final int INVALID = -1; // Invalid values for when a
requirement is not met

    /**
     * Map to represent the machine op table.
     */
    private HashMap<String, Pseudo_Op_Info> table;
```

```
    /**
     * Parser specialized to deal with parsing immediates in the assembly
language.
     */
    private String_Parser sp;

    /**
     * Default constructor.
     */
    public Pseudo_Op_Table() {
        this.table = new HashMap<>();
        sp = new String_Parser();
        loadTable();
    }

    /**
     * Loads the 6 Pseudo-Ops for the assembler to the Pseudo-Ops table,
which
     * contains the mnemonic name, size defined by the pseudo-ops
behavior.
     *
     * @ensures Pseudo_Op_Table = <"pseudo-op name", "size", <definite
length,
     *          variable length>>
     */
    public void loadTable() {
        // Ops with size 0
        // Regular ops
        this.table.put(".ORIG", new
Pseudo_Op_Info(Pseudo_Op_Ins_Enum.ORIG, ZERO, Pseudo_Op_Format.DEFINITE));
        this.table.put(".EQU", new Pseudo_Op_Info(Pseudo_Op_Ins_Enum.EQU,
ZERO, Pseudo_Op_Format.DEFINITE));
        this.table.put(".END", new Pseudo_Op_Info(Pseudo_Op_Ins_Enum.END,
ZERO, Pseudo_Op_Format.DEFINITE));

        // Linking/Loader ops
        this.table.put(".ENT", new Pseudo_Op_Info(Pseudo_Op_Ins_Enum.ENT,
ZERO, Pseudo_Op_Format.DEFINITE));
```

```
        this.table.put(".EXT", new Pseudo_Op_Info(Pseudo_Op_Ins_Enum.EXT,
ZERO, Pseudo_Op_Format.DEFINITE));

        // Ops with size 1
        this.table.put(".FILL", new
Pseudo_Op_Info(Pseudo_Op_Ins_Enum.FILL, ONE, Pseudo_Op_Format.DEFINITE));

        // Ops with variable size length
        this.table.put(".BLKW", new
Pseudo_Op_Info(Pseudo_Op_Ins_Enum.BLKW, ZERO, Pseudo_Op_Format.VARIABLE));
        this.table.put(".STRZ", new
Pseudo_Op_Info(Pseudo_Op_Ins_Enum.STRZ, ZERO, Pseudo_Op_Format.VARIABLE));
    }

    /**
     * Returns if the given instruction is in the Pseudo-Ops Table
     *
     * @param key Mnemonic name for the instruction
     * @return the boolean value for if the given instruction is in the
Pseudo-Ops
     *         Table or not
     * @ensures containsOp = Pseudo_Op_Table contains key (true), or false
     */
    public boolean containsOp(String key) {
        return this.table.containsKey(key);
    }

    /**
     * Returns the enumerated version of the instruction name, or null if
it is
     * invalid.
     *
     * @param key Mnemonic name for the instruction
     * @return enumerated version of the key or null for invalid
instructions
     * @ensures getInstructionName = enum_name(key) or null
     */
    public Pseudo_Op_Ins_Enum getInstructionName(String key) {
        if (!this.table.containsKey(key)) {
            return null;
```

```
        }

        Pseudo_Op_Info op = this.table.get(key);
        return op.getInstructionName();
    }

    /**
     * Returns the word size for the instruction, or -1 if the instruction
is
     * invalid.
     *
     * @param key Mnemonic name for the instruction
     * @return word size of the key instruction or -1 if the instruction
is invalid
     * @ensures getLength = word_size(key) or -1
     */
    public int getLength(String key) {
        if (!this.table.containsKey(key)) {
            return INVALID;
        }

        Pseudo_Op_Info op = this.table.get(key);
        return op.getLength();
    }

    /**
     * Returns the format of the instruction, or null if the instruction
is invalid.
     *
     * @param key Mnemonic name for the instruction
     * @return format of the key instruction or null if the instruction is
invalid
     * @ensures getFormat = format(key) or null
     */
    public Pseudo_Op_Format getFormat(String key) {
        if (!this.table.containsKey(key)) {
            return null;
        }

        Pseudo_Op_Info op = this.table.get(key);
```

```
        return op.getFormat();
    }

    /**
     * Returns the size for the block of words to be allocated based on
the operand
     * size.
     *
     * @param str operand that contains the numerical value of the block
of words
     * @return the number of words to allocate based on the operand, -1 if
not valid
     * @ensures blockLength = numerical_length(str) or -1
     */
    public int blockLength(String str) {
        // If the block can be parsed
        if (!sp.canParseInt(str)) {
            return INVALID;
        }

        int val = sp.parseAddress(str);

        // If the block value is <= 0
        if (val < ONE) {
            return INVALID;
        }

        return val;
    }
}
```

Passes Package

Pass1.java

```
package lab3_integrated.assembler.Passes;

import java.util.ArrayList;
```

```
import java.util.HashMap;
import java.util.List;

public class Pass1 {

    private static ArrayList<String> LitArray = new ArrayList<String>();
    private static ArrayList<String> ExtSyms = new ArrayList<String>();

    /**
     * Calls methods to fill both the symbol table and literal table
     *
     * @param arr      The array containing all the lines of the file that
were
     *                 parsed down in the validator
     * @param loc      Initial location counter to be used throughout pass
1
     * @param SymTable Empty symbol table to be filled
     * @param LitTable Empty literal table to be filled
     *
     */
    public static int fillTables(ArrayList<List<String>> arr, int loc,
HashMap<String, String[]> SymTable,
            HashMap<String, Integer> LitTable) {
        loc = fillSymTable(arr, SymTable, loc);
        loc = fillLitTable(LitTable, loc, LitArray);
        return loc;
    }

    /**
     * Loops through the array of lines to fill the symbol table
     *
     * @param arr      The array of lines to be looped through
     * @param SymTable Empty symbol table to be filled
     * @param loc      The current location counter
     *
     */
    public static int fillSymTable(ArrayList<List<String>> arr,
HashMap<String, String[]> SymTable, int loc) {
        String sym, instruct, operand;
        for (int i = 1; i < arr.size() - 1; i++) {
```

```
            if (SymTable.size() > 100) {
                System.out.println("Too many symbols");
                System.exit(1);
            }
            // Assigns values for the symbol, instruction and operand from
the current line
            sym = arr.get(i).get(0);
            instruct = arr.get(i).get(1);
            operand = arr.get(i).get(2);

            // Fill the symbols array that will become the value for the
HashMap
            String[] symArr;
            if (!sym.equals("") && !SymTable.containsKey(sym)) {
                symArr = fillSymArr(sym, instruct, operand, SymTable,
loc);
                SymTable.put(sym, symArr);
            } else if (instruct.equals(".EXT")) {
                String[] splitOperand = operand.split(",");
                for (int j = 0; j < splitOperand.length; j++) {
                    ExtSyms.add(splitOperand[j]);
                }
            }

            // Add any literals in the line to an ArrayList to later fill
the literal table
            if (operand.contains("=") && !instruct.equals(".STRZ")) {
                String lit = operand.substring(operand.indexOf("="));
                LitArray.add(lit);
            }

            // Increment the location counter
            loc = incLoc(instruct, operand, loc);
        }
        return loc;
    }

    /**
     * Loops through the array of stored literal values and inputs them
into the
```

```
     * literal table at the correct location
     *
     * @param LitTable Empty literal table to be filled
     * @param locCount The current location counter
     * @param litArray The array of literals to fill the literal table
with
     *
     */
    public static int fillLitTable(HashMap<String, Integer> LitTable, int
locCount, ArrayList<String> litArray) {
        for (int i = 0; i < litArray.size(); i++) {
            if (LitTable.size() > 50) {
                System.out.println("Too many literals");
                System.exit(1);
            }
            LitTable.put(litArray.get(i), locCount);
            locCount++;
            locCount %= 0x10000;
        }
        return locCount;
    }

    /**
     * Fills array for a symbol containing the location of the symbol and
whether it
     * is relative or absolute
     *
     * @param sym      String containing the symbol
     * @param instruct Instruction pertaining to the symbol
     * @param operand  The operand pertaining to the symbol
     * @param SymTable SymTable in case there needs to be a check for an
absolute
     *                 symbol
     * @param locCount The current location counter
     *
     */
    public static String[] fillSymArr(String sym, String instruct, String
operand, HashMap<String, String[]> SymTable,
            int locCount) {
        String[] symArr = new String[2];
```

```
        if (!instruct.equals(".EQU") && !instruct.equals(".EXT")) {
            symArr[0] = "x" + Integer.toHexString(locCount);
            symArr[1] = "R";
        } else {
            symArr = getEQUVal(sym, operand, SymTable);
        }
        return symArr;

    }

    /**
     * Gets the final value of any EQU instructions
     *
     * @param sym      Symbol to check the value of
     * @param operand  Operand containing either the value of the .EQU
symbol or
     *                 another symbol to check the value of
     * @param SymTable Symbol table to reference if the operand is another
symbol
     */
    public static String[] getEQUVal(String sym, String operand,
HashMap<String, String[]> SymTable) {
        String[] val = new String[2];

        if (operand.charAt(0) == '#' || operand.charAt(0) == 'x') {
            // If the operand is a single value
            val[0] = operand;
            val[1] = "A";
        } else {
            // If the operand is a symbol or a series of operands
            if (!SymTable.containsKey(operand) &&
!ExtSyms.contains(operand)) {
                System.out.println("Forward referencing error");
                System.exit(1);
            } else if (SymTable.containsKey(operand)) {
                String[] symArr = SymTable.get(operand);
                if (symArr[1].equals("A")) {
                    val[1] = "A";
                } else {
                    val[1] = "R";
```

```
                }
                val[0] = symArr[0];
            } else {
                val[0] = operand;
                val[1] = "R";
            }

        }

        return val;
    }

    /**
     * Increments the location counter according to which instruction is
being
     * checked
     *
     * @param instruct Instruction to be checked for how much to increment
the
     *                 location counter
     * @param operand  Operand to be parsed to see how mcuh to increment
the
     *                 location counter
     * @param locCount The current location counter
     *
     */
    public static int incLoc(String instruct, String operand, int
locCount) {
        if (instruct.equals(".STRZ")) {
            locCount += operand.length() + 1;
        } else if (instruct.equals(".BLKW")) {
            if (operand.charAt(0) == 'x') {
                locCount += Integer.parseInt(operand.substring(1), 16);
            } else {
                locCount += Integer.parseInt(operand.substring(1));
            }
        } else if (!instruct.equals(".EQU") && !instruct.equals(".ENT") &&
!instruct.equals(".EXT")) {
            locCount++;
        }
```

```
        locCount %= 0x10000;
        return locCount;
    }
}
```

Pass2.java

```
package lab3_integrated.assembler.Passes;

import java.io.FileWriter;
import java.io.IOException;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

import lab3_integrated.assembler.MOT.Machine_Op_Ins_Enum;
import lab3_integrated.assembler.MOT.Machine_Op_Table;
import lab3_integrated.assembler.POT.Pseudo_Op_Format;
import lab3_integrated.assembler.POT.Pseudo_Op_Ins_Enum;
import lab3_integrated.assembler.POT.Pseudo_Op_Table;
import lab3_integrated.assembler.lab2.Exceptions.Pass2Exception;
import lab3_integrated.assembler.lab2.Validate.Locations;

public class Pass2 {

    /**
     * Location counter for tracking the address of the current assembly
line.
     */
    private int LC;

    /**
     * Machine-Ops table containing each machine instruction.
     */
    private Machine_Op_Table mot;
```

```
    /**
     * Pseudo-Ops table containing each assembler instructions.
     */
    private Pseudo_Op_Table pot;

    /**
     * Writing the contents of the object file.
     */
    private FileWriter outFile;

    /**
     * Writing the contents of the listing file.
     */
    private FileWriter listingFile;

    /**
     * Parser for parsing strings into integer values.
     */
    private String_Parser sp;

    /**
     * The size of the object file.
     */
    private int segmentSize;

    /**
     * Instance of the symbol table.
     */
    private Map<String, String[]> symbolTable;

    /**
     * Instance of the literal table.
     */
    private Map<String, Integer> literalTable;

    /**
     * Flag to indicate whether a program is relocatable or not.
     */
    private boolean isRelocatable;
```

```
    /**
     * The initial load address.
     */
    private String initialAddress;

    /**
     * Temporary external symbol table to contain the external symbols.
     */
    private Map<String, String> tempExternalTable;

    /**
     * Storing the name of the segment.
     */
    private String segName;

    /**
     * Default constructor for the pass 2 class.
     *
     * @param mot          machine ops table
     * @param pot          pseudo ops table
     * @param symbolTable  symbol table
     * @param literalTable literal table
     * @param objectFile   file writer for the object file
     * @param listingFile  file writer for the listing file
     * @param segmentSize  the total size of the segment
     */
    public Pass2(Machine_Op_Table mot, Pseudo_Op_Table pot, Map<String,
String[]> symbolTable,
            Map<String, Integer> literalTable, FileWriter objectFile,
FileWriter listingFile, int segmentSize) {
        // Initializing the tables
        this.mot = mot;
        this.pot = pot;

        this.symbolTable = symbolTable;
        this.literalTable = literalTable;
        this.tempExternalTable = new HashMap<>();

        // Other elements of the pass 2
        this.LC = 0;
```

```
        this.sp = new String_Parser();
        this.segmentSize = segmentSize;
        this.isRelocatable = false;
        this.initialAddress = "0000";
        this.segName = "";

        // Trying to create the object file and listing file
        this.outFile = objectFile;
        this.listingFile = listingFile;
    }

    /**
     * Main method for parsing through each line of assembly code and
displaying to
     * the object file and listing file.
     *
     * @param list List containing each line of assembly to be parsed,
without the
     *             comments or junk value
     * @throws Pass2Exception throwing a runtime error when a fatal error
is
     *                        detected in Pass2
     */
    public void parseInput(List<List<String>> list) throws Pass2Exception
{
        for (int i = 0; i < list.size(); i++) {
            // Getting the four parts of the assembly code
            String symbol = list.get(i).get(Locations.LABEL);
            String opcode = list.get(i).get(Locations.OPERATION);
            String operand = list.get(i).get(Locations.OPERANDS);
            String line = list.get(i).get(Locations.LINE);

            // Adding any literals to the object file (before .END)
            if (i == list.size() - 1) {
                for (Map.Entry<String, Integer> entry :
literalTable.entrySet()) {
                    String textRecord = literalObjectFile(entry.getKey(),
entry.getValue(), line);

                    // Writing the contents to the output file
```

```
                    try {
                        this.outFile.write(textRecord);
                    } catch (IOException e) {
                        throw new Pass2Exception(
                                "Line " + line + ": Unable to add assembly
line code to the object file");

                    }
                }
            }

            // Opcode defined in the Pseudo_Ops_Table
            String assembleLine;
            if (pot.containsOp(opcode)) {
                assembleLine = assemblePseudoOpLine(symbol, opcode,
operand, line);
            }

            // Opcode contained in the Machine_Ops_Table
            else {
                assembleLine = assembleMachineOpLine(opcode, operand,
line);
            }

            // Writing the contents to the output file
            try {
                this.outFile.write(assembleLine);
            } catch (IOException e) {
                throw new Pass2Exception("Line " + line + ": Unable to add
assembly line code to the object file");

            }

            String listFileOutput = outputToListingFile(symbol, opcode,
operand, assembleLine, line);

            // Writing the contents to the listing file
            try {
                this.listingFile.write(listFileOutput);
            } catch (IOException e) {
```

```
                throw new Pass2Exception("Line " + line + ": Unable to add
assembly line code to the listing file");

            }

            // Adding any literals to the listing file (after .END)
            if (i == list.size() - 1) {
                for (Map.Entry<String, Integer> entry :
literalTable.entrySet()) {
                    String listFile = literalListingFile(entry.getKey(),
entry.getValue(), line);

                    // Writing the contents to the listing file
                    try {
                        this.listingFile.write(listFile);
                    } catch (IOException e) {
                        throw new Pass2Exception(
                                "Line " + line + ": Unable to add assembly
line code to the listing file");

                    }
                }
            }
        }

        // Closing the output and listing file streams
        try {
            this.outFile.close();
        } catch (IOException e) {
            throw new Pass2Exception("Unable to close object file");

        }

        try {
            this.listingFile.close();
        } catch (IOException e) {
            throw new Pass2Exception("Unable to close listing file");

        }
    }
```

```
    /**
     * Creating the text records for the literal table before the .END
operation
     * executes.
     *
     * @param literal the literal value to be added to text records
     * @param value   the corresponding address for a given literal
     * @param line    the current line of assembly code
     * @return returns the text record string
     * @throws Pass2Exception throwing a runtime error when a fatal error
is
     *                        detected in Pass2
     */
    public String literalObjectFile(String literal, Integer value, String
line) throws Pass2Exception {
        String textRecord = "T";

        String literalAddress = sp.literalValue(literalTable, literal,
line);
        textRecord += literalAddress.substring(1);

        String litVal = sp.parseLiteralString(literal);

        if (litVal == null) {
            throw new Pass2Exception("Line " + line + ": Invalid literal
value");
        }

        textRecord += litVal + "\n";
        return textRecord;
    }

    /**
     * Creating the text records for the literal table before the .END
operation
     * executes.
     *
     * @param literal the literal value to be added to text records
     * @param value   the corresponding address for a given literal
```

```
     * @param line    the current line of assembly code
     * @return returns the listing file record string
     * @throws Pass2Exception throwing a runtime error when a fatal error
is
     *                        detected in Pass2
     */
    public String literalListingFile(String literal, Integer value, String
line) throws Pass2Exception {
        String assemblyLine = literalObjectFile(literal, value, line);

        // Address value
        String address = assemblyLine.substring(1, 5);
        String listFileOutput = "(" + address + ")";

        // Hex value
        String hexVal = assemblyLine.substring(5, 9);
        listFileOutput += (" " + hexVal);

        // Binary value
        int hexInt = Integer.parseInt(hexVal, 16);
        String binVal = Integer.toBinaryString(hexInt);
        while (binVal.length() < 16) {
            binVal = "0" + binVal;
        }

        listFileOutput += (" " + binVal + " (");
        listFileOutput += " lit) \n";

        return listFileOutput;
    }

    /**
     * Going through the possible machine instructions and creating the
text
     * records.
     *
     * @param opcode  the operation to execute for the line of assembly
     * @param operand the operands of the instruction
     * @param line    the assembly line number
```

```
     * @return the text record with the location, instruction, and
modification
     *         record (if applicable)
     * @throws Pass2Exception throwing a runtime error when a fatal error
is
     *                        detected in Pass2
     */
    public String assembleMachineOpLine(String opcode, String operand,
String line) throws Pass2Exception {
        // Getting the enumerated name of the instruction, the value, and
the size
        Machine_Op_Ins_Enum ins_name = mot.getInstructionName(opcode);
        int value = mot.getOpcode(opcode);
        int size = mot.getSize(opcode);

        // Splitting the operands by the elements they have
        String[] splitOperand = operand.split(",");
        String record = "T";

        // Using a switch-case block for each of the machine instructions
present.
        switch (ins_name) {
        case ADD:
            record += addOp(splitOperand, value, size, line) + "\n";
            break;
        case AND:
            record += andOp(splitOperand, value, size, line) + "\n";
            break;
        case BR:
            record += brxOp(splitOperand, "0", "0", "0", value, size,
line) + "\n";
            break;
        case BRN:
            record += brxOp(splitOperand, "1", "0", "0", value, size,
line) + "\n";
            break;
        case BRZ:
            record += brxOp(splitOperand, "0", "1", "0", value, size,
line) + "\n";
            break;
```

```
        case BRP:
            record += brxOp(splitOperand, "0", "0", "1", value, size,
line) + "\n";
            break;
        case BRNZ:
            record += brxOp(splitOperand, "1", "1", "0", value, size,
line) + "\n";
            break;
        case BRNP:
            record += brxOp(splitOperand, "1", "0", "1", value, size,
line) + "\n";
            break;
        case BRZP:
            record += brxOp(splitOperand, "0", "1", "1", value, size,
line) + "\n";
            break;
        case BRNZP:
            record += brxOp(splitOperand, "1", "1", "1", value, size,
line) + "\n";
            break;
        case DBUG:
            record += dbugOp(value, size, line) + "\n";
            break;
        case JSR:
            record += jsrOp(splitOperand, "1", value, size, line) + "\n";
            break;
        case JMP:
            record += jsrOp(splitOperand, "0", value, size, line) + "\n";
            break;
        case JSRR:
            record += jsrrOp(splitOperand, "1", value, size, line) + "\n";
            break;
        case JMPR:
            record += jsrrOp(splitOperand, "0", value, size, line) + "\n";
            break;
        case LD:
            record += ldOp(splitOperand, value, size, line) + "\n";
            break;
        case LDI:
            record += ldiOp(splitOperand, value, size, line) + "\n";
```

```
            break;
        case LDR:
            record += ldrOp(splitOperand, value, size, line) + "\n";
            break;
        case LEA:
            record += leaOp(splitOperand, value, size, line) + "\n";
            break;
        case NOT:
            record += notOp(splitOperand, value, size, line) + "\n";
            break;
        case RET:
            record += retOp(value, size, line) + "\n";
            break;
        case ST:
            record += stOp(splitOperand, value, size, line) + "\n";
            break;
        case STI:
            record += stiOp(splitOperand, value, size, line) + "\n";
            break;
        case STR:
            record += strOp(splitOperand, value, size, line) + "\n";
            break;
        case TRAP:
            record += trapOp(splitOperand, value, size, line) + "\n";
            break;
        default:
            // Error?
            break;
        }

        return record;
    }

    /**
     * Going through the possible psueodo ops instructions and creating
the text
     * records if necessary.
     *
     *
     * @param symbol  any symbol if present in the assembly line
```

```
     * @param opcode  the operation to execute for the line of assembly
     * @param operand the operands of the instruction
     * @param line    the assembly line number
     * @return the text record (if applicable) with the location,
instruction, and
     *         modification record (if applicable)
     * @throws Pass2Exception throwing a runtime error when a fatal error
is
     *                        detected in Pass2
     */
    public String assemblePseudoOpLine(String symbol, String opcode,
String operand, String line)
            throws Pass2Exception {
        // Getting the enumerated name of the instruction
        Pseudo_Op_Ins_Enum ins_name = pot.getInstructionName(opcode);

        // Splitting the operands by the elements they have (for .ENT and
.EXT)
        String[] splitOperand = operand.split(",");
        String record = "";

        // Using a switch-case block for each of the assembler
instructions present.
        switch (ins_name) {
        case BLKW:
            record += blkwOp(symbol, opcode, operand, line);
            break;
        case END:
            record += endOp(operand, line);
            break;
        case ENT:
            record += entOp(splitOperand, line);
            break;
        case EXT:
            record += extOp(splitOperand, line);
            break;
        case EQU:
            record += equOp(operand, line);
            break;
        case FILL:
```

```
            record += fillOp(operand, line);
            break;
        case ORIG:
            record += origOp(symbol, operand, line);
            break;
        case STRZ:
            record += strzOp(symbol, opcode, operand, line);
            break;
        default:
            // Error?
            break;
        }

        return record;
    }

    /**
     * Parsing the line of assembly code into a listing file.
     *
     * @param symbol       any symbol if present in the assembly line
     * @param opcode       the operation to execute for the line of
assembly
     * @param operand      the operands of the instruction
     * @param assemblyLine the line of assembly code that contains the
text record
     * @param line         the assembly line number
     * @return the string following the format for the listing file
     */
    public String outputToListingFile(String symbol, String opcode, String
operand, String assemblyLine, String line) {
        String listFile = "";

        if (this.pot.containsOp(opcode) && this.pot.getFormat(opcode) ==
Pseudo_Op_Format.DEFINITE
                && this.pot.getLength(opcode) == 0) {
            // Account for record that don't occupy memory
            listFile = "\t\t\t\t\t\t\t (";

            // Line number
            while (line.length() < 4) {
```

```
                line = " " + line;
            }

            // Symbol value
            while (symbol.length() < 16) {
                symbol += " ";
            }

            listFile += (line + ") " + symbol);

            // Opcode and operand
            while (opcode.length() < 6) {
                opcode += " ";
            }

            listFile += (opcode + operand + "\n");
        }

        else if (this.mot.containsOp(opcode) ||
(this.pot.containsOp(opcode) && this.pot.getLength(opcode) == 1)) {
            // Address value
            String address = assemblyLine.substring(1, 5);
            listFile = "(" + address + ")";

            // Hex value
            String hexVal = assemblyLine.substring(5, 9);
            listFile += (" " + hexVal);

            // Binary value
            int hexInt = Integer.parseInt(hexVal, 16);
            String binVal = Integer.toBinaryString(hexInt);
            while (binVal.length() < 16) {
                binVal = "0" + binVal;
            }

            listFile += (" " + binVal + " (");

            // Line number
            while (line.length() < 4) {
                line = " " + line;
```

```
            }

            // Symbol value
            while (symbol.length() < 16) {
                symbol += " ";
            }

            listFile += (line + ") " + symbol);

            // Opcode and operand
            while (opcode.length() < 6) {
                opcode += " ";
            }

            listFile += (opcode + operand + "\n");
        }

        return listFile;
    }

    /**
     * Parsing the line of .BLKW op into a listing file.
     *
     * @param symbol  any symbol if present in the assembly line
     * @param address the address specified by the location counter
     * @param opcode  the operation to execute for the line of assembly
     * @param operand the operands of the instruction
     * @param line    the assembly line number
     * @throws Pass2Exception throwing a runtime error when a fatal error
is
     *                        detected in Pass2
     */
    public void blkwListingFile(String symbol, String address, String
opcode, String operand, String line)
            throws Pass2Exception {
        // Writing the contents to the listing file
        String listFileOutput = "(" + address + ")";
        listFileOutput += " \t \t\t\t\t\t (";

        // Line number
```

```
        while (line.length() < 4) {
            line = " " + line;
        }

        // Symbol value
        while (symbol.length() < 16) {
            symbol += " ";
        }

        listFileOutput += (line + ") " + symbol);

        // Opcode and operand
        while (opcode.length() < 6) {
            opcode += " ";
        }

        listFileOutput += (opcode + operand + "\n");

        try {
            this.listingFile.write(listFileOutput);
        } catch (IOException e) {
            throw new Pass2Exception("Line " + line + ": Unable to add
assembly line code to the listing file");
        }
    }

    /**
     * Parsing the line of .STRZ op into a listing file.
     *
     * @param symbol       any symbol if present in the assembly line
     * @param opcode       the operation to execute for the line of
assembly
     * @param operand      the operands of the instruction
     * @param i            index to determine if the operand should be
printed or
     *                     not
     * @param assemblyLine the line of assembly code that contains the
text record
     * @param line         the assembly line number
```

```
     * @throws Pass2Exception throwing a runtime error when a fatal error
is
     *                        detected in Pass2
     */
    public void strzListingFile(String symbol, String opcode, String
operand, int i, String assemblyLine, String line)
            throws Pass2Exception {
        // Address value
        String address = assemblyLine.substring(1, 5);
        String listFileOutput = "(" + address + ")";

        // Hex value
        String hexVal = assemblyLine.substring(5, 9);
        listFileOutput += (" " + hexVal);

        // Binary value
        int hexInt = Integer.parseInt(hexVal, 16);
        String binVal = Integer.toBinaryString(hexInt);
        while (binVal.length() < 16) {
            binVal = "0" + binVal;
        }

        listFileOutput += (" " + binVal + " (");

        // Line number
        while (line.length() < 4) {
            line = " " + line;
        }

        listFileOutput += line + ") ";

        if (i == 0) {
            // Symbol value
            while (symbol.length() < 16) {
                symbol += " ";
            }

            listFileOutput += symbol;

            // Opcode and operand
```

```
            while (opcode.length() < 6) {
                opcode += " ";
            }

            listFileOutput += (opcode + "\"" + operand + "\"\n");
        } else {
            listFileOutput += "\n";
        }

        // Writing the contents to the listing file
        try {
            this.listingFile.write(listFileOutput);
        } catch (IOException e) {
            throw new Pass2Exception("Line " + line + ": Unable to add
assembly line code to the listing file");
        }
    }

    /**
     * Forming the text record for the ADD instruction.
     *
     * @param operand      each of the operand split into a separate
string
     * @param opcode_value the size by which LC is to be updated
     * @param size         the size by which LC is to be updated
     * @param line         the current line in the assembly code being
parsed
     * @return the text record for the ADD instruction
     * @throws Pass2Exception throwing a runtime error when a fatal error
is
     *                        detected in Pass2
     */
    private String addOp(String[] operand, int opcode_value, int size,
String line) throws Pass2Exception {
        String textRecord = "";

        // Forming the address based on location counter
        String lcAddress = addressFromLC(size);
        textRecord += lcAddress;
```

```
        String binaryString = sp.opcodeBinaryString(opcode_value);

        // Iterating through the operand length
        for (int i = 0; i < operand.length; i++) {
            String pos = operand[i];

            // If the operand is a register
            if (pos.charAt(0) == 'R') {
                // Adding the don't care values if the operand only
involved registers
                if (i == 2) {
                    binaryString += "000";
                }

                // Convert the register value to fit the binary
                String regVal = sp.registerToBinaryString("#" +
pos.substring(1), line);
                binaryString += regVal;
            }

            // If the operand is an immediate
            else if (pos.charAt(0) == '#' || pos.charAt(0) == 'x') {
                binaryString += "1";
                String immVal = sp.immediateToBinaryString(pos, line);
                binaryString += immVal;
            }

            // If the operand is an absolute symbol
            else {
                // Ensure the symbol is in the symbol table
                absoluteSymbolTable(pos, line);

                // If there is a symbol at the end, it is treated as an
immediate
                if (i == 2) {
                    binaryString += "1";
                    String immVal =
sp.immediateToBinaryString(symbolTable.get(pos)[0], line);
                    binaryString += immVal;
                }
```

```
                // Otherwise, treat it is a register
                else {
                    // Convert the register value to fit the binary
                    String regVal =
sp.registerToBinaryString(symbolTable.get(pos)[0], line);
                    binaryString += regVal;
                }
            }
        }

        // Convert the binary string into a hex string
        String hexString = sp.convertBinaryToHexString(binaryString,
line);
        textRecord += hexString;

        return textRecord;
    }

    /**
     * Forming the text record for the AND instruction.
     *
     * @param operand      each of the operand split into a separate
string
     * @param opcode_value the size by which LC is to be updated
     * @param size         the size by which LC is to be updated
     * @param line         the current line in the assembly code being
parsed
     * @return the text record for the AND instruction
     * @throws Pass2Exception throwing a runtime error when a fatal error
is
     *                        detected in Pass2
     */
    private String andOp(String[] operand, int opcode_value, int size,
String line) throws Pass2Exception {
        // Calling the instruction with the same procedure
        return addOp(operand, opcode_value, size, line);
    }

    /**
```

```
     * Forming the text record for the BRx instruction.
     *
     * @param operand      each of the operand split into a separate
string
     * @param n            the N (negative) bit
     * @param z            the Z (zero) bit
     * @param p            the P (positive) bit
     * @param opcode_value the size by which LC is to be updated
     * @param size         the size by which LC is to be updated
     * @param line         the current line in the assembly code being
parsed
     * @return the text record for the BRx instruction
     * @throws Pass2Exception throwing a runtime error when a fatal error
is
     *                        detected in Pass2
     */
    private String brxOp(String[] operand, String n, String z, String p,
int opcode_value, int size, String line)
            throws Pass2Exception {
        String textRecord = "";

        // Forming the address based on location counter
        String lcAddress = addressFromLC(size);
        textRecord += lcAddress;

        String binaryString = sp.opcodeBinaryString(opcode_value);

        // Adding the N, Z, P bits to the binary string
        binaryString += (n + z + p);

        String address = operand[0];

        int relative = 0;

        // name of the external symbol for the modification record
        String externalSym = address;

        // If the given address is a symbol
        if (!(address.charAt(0) == '#' || address.charAt(0) == 'x')) {
            /*
```

```
            * Ensure the symbol is present in the table and check if the
symbol is absolute
            * or relative
            */
            relative = isRelativeSymbol(address, line);

            // Check address if it is a .EXT symbol or from symbol table
            if (relative == 2) {
                address = this.tempExternalTable.get(address);
            } else {
                String val = this.symbolTable.get(address)[0];

                // Check if the address is a value
                if (val.charAt(0) == 'x' || val.charAt(0) == '#') {
                    address = val;
                }

                // otherwise, it is a external symbol
                else {

                    // make sure the symbol is a external forward
referencing
                    if (!this.tempExternalTable.containsKey(val)) {
                        throw new Pass2Exception("Line: " + line + ":
cannot forward reference a non-external symbol");
                    }

                    // update the value with the value from the external
symbol table
                    address = this.tempExternalTable.get(val);
                    externalSym = val;

                    // when creating modification records, make sure it is
a external symbol
                    relative = 2;
                }
            }
        }

        /*
```

```
         * Check if the given address is within page range (with the
location counter)
         * and if so, return the lower 9 bits of the given address
         */
        String offset = sp.pageRangeAndOffset(lcAddress, address, line);
        binaryString += offset;

        // Convert the binary string into a hex string
        String hexString = sp.convertBinaryToHexString(binaryString,
line);
        textRecord += hexString;

        // Check if the symbol was a relative symbol to add a modification
record
        if (this.isRelocatable) {
            if (relative == 1) {
                textRecord += ("X9" + this.segName);
            } else if (relative == 2) {
                textRecord += ("X9" + externalSym);
            }
        }

        return textRecord;
    }

    /**
     * Forming the text record for the DBUG instruction.
     *
     * @param opcode_value the size by which LC is to be updated
     * @param size         the size by which LC is to be updated
     * @param line         the current line in the assembly code being
parsed
     * @return the text record for the DBUG instruction
     */
    private String dbugOp(int opcode_value, int size, String line) {
        String textRecord = "";

        // Forming the address based on location counter
        String lcAddress = addressFromLC(size);
        textRecord += lcAddress;
```

```
        String binaryString = sp.opcodeBinaryString(opcode_value);

        // Adding the don't cares
        binaryString += "000000000000";

        // Convert the binary string into a hex string
        String hexString = sp.convertBinaryToHexString(binaryString,
line);
        textRecord += hexString;

        return textRecord;
    }

    /**
     * Forming the text record for the JSR/JMP instruction.
     *
     * @param operand      each of the operand split into a separate
string
     * @param L            the bit to see if the address is to be saved
     * @param opcode_value the size by which LC is to be updated
     * @param size         the size by which LC is to be updated
     * @param line         the current line in the assembly code being
parsed
     * @return the text record for the JSR/JMP instruction
     * @throws Pass2Exception throwing a runtime error when a fatal error
is
     *                        detected in Pass2
     */
    private String jsrOp(String[] operand, String L, int opcode_value, int
size, String line) throws Pass2Exception {
        String textRecord = "";

        // Forming the address based on location counter
        String lcAddress = addressFromLC(size);
        textRecord += lcAddress;

        String binaryString = sp.opcodeBinaryString(opcode_value);

        // Adding the L bits and don't cares to the binary string
```

```
        binaryString += (L + "00");

        String address = operand[0];

        int relative = 0;

        // name of the external symbol for the modification record
        String externalSym = address;

        // If the given address is a symbol
        if (!(address.charAt(0) == '#' || address.charAt(0) == 'x')) {
            /*
             * Ensure the symbol is present in the table and check if the
symbol is absolute
             * or relative
             */
            relative = isRelativeSymbol(address, line);

            // Update address if it is a .EXT symbol
            if (relative == 2) {
                address = this.tempExternalTable.get(address);
            } else {
                String val = this.symbolTable.get(address)[0];

                // Check if the address is a value
                if (val.charAt(0) == 'x' || val.charAt(0) == '#') {
                    address = val;
                }

                // otherwise, it is a external symbol
                else {

                    // make sure the symbol is a external forward
referencing
                    if (!this.tempExternalTable.containsKey(val)) {
                        throw new Pass2Exception("Line: " + line + ":
cannot forward reference a non-external symbol");
                    }
```

```
                    // update the value with the value from the external
symbol table
                    address = this.tempExternalTable.get(val);
                    externalSym = val;

                    // when creating modification records, make sure it is
a external symbol
                    relative = 2;
                }
            }
        }

        /*
         * Check if the given address is within page range (with the
location counter)
         * and if so, return the lower 9 bits of the given address
         */
        String offset = sp.pageRangeAndOffset(lcAddress, address, line);
        binaryString += offset;

        // Convert the binary string into a hex string
        String hexString = sp.convertBinaryToHexString(binaryString,
line);
        textRecord += hexString;

        // Check if the symbol was a relative symbol to add a modification
record
        if (this.isRelocatable) {
            if (relative == 1) {
                textRecord += ("X9" + this.segName);
            } else if (relative == 2) {
                textRecord += ("X9" + externalSym);
            }
        }

        return textRecord;
    }

    /**
     * Forming the text record for the JSRR/JMPR instruction.
```

```
     *
     * @param operand      each of the operand split into a separate
string
     * @param L            the bit to see if the address is to be saved
     * @param opcode_value the size by which LC is to be updated
     * @param size         the size by which LC is to be updated
     * @param line         the current line in the assembly code being
parsed
     * @return the text record for the JSRR/JMPR instruction
     * @throws Pass2Exception throwing a runtime error when a fatal error
is
     *                        detected in Pass2
     */
    private String jsrrOp(String[] operand, String L, int opcode_value,
int size, String line) throws Pass2Exception {
        String textRecord = "";

        // Forming the address based on location counter
        String lcAddress = addressFromLC(size);
        textRecord += lcAddress;

        String binaryString = sp.opcodeBinaryString(opcode_value);

        // Adding the L bit and the don't cares
        binaryString += (L + "00");

        // Iterating through the operand length
        for (int i = 0; i < operand.length; i++) {
            String pos = operand[i];

            // If the operand is a register
            if (pos.charAt(0) == 'R') {
                // Convert the register value to fit the binary
                String regVal = sp.registerToBinaryString("#" +
pos.substring(1), line);
                binaryString += regVal;
            }

            // If the operand is an index
            else if (pos.charAt(0) == '#' || pos.charAt(0) == 'x') {
```

```
                String indVal = sp.indexToBinaryString(pos, line);
                binaryString += indVal;
            }

            // If the operand is an absolute symbol
            else {
                // Ensure the symbol is in the symbol table
                absoluteSymbolTable(pos, line);

                // If there is a symbol at the end, it is treated as an
index
                if (i == 1) {
                    String indVal =
sp.indexToBinaryString(symbolTable.get(pos)[0], line);
                    binaryString += indVal;
                }

                // Otherwise, treat it is a register
                else {
                    // Convert the register value to fit the binary
                    String regVal =
sp.registerToBinaryString(symbolTable.get(pos)[0], line);
                    binaryString += regVal;
                }
            }
        }

        // Convert the binary string into a hex string
        String hexString = sp.convertBinaryToHexString(binaryString,
line);
        textRecord += hexString;

        return textRecord;
    }

    /**
     * Forming the text record for the LD instruction.
     *
     * @param operand      each of the operand split into a separate
string
```

```
     * @param opcode_value the size by which LC is to be updated
     * @param size         the size by which LC is to be updated
     * @param line         the current line in the assembly code being
parsed
     * @return the text record for the LD instruction
     * @throws Pass2Exception throwing a runtime error when a fatal error
is
     *                        detected in Pass2
     */
    private String ldOp(String[] operand, int opcode_value, int size,
String line) throws Pass2Exception {
        String textRecord = "";

        // Forming the address based on location counter
        String lcAddress = addressFromLC(size);
        textRecord += lcAddress;

        String binaryString = sp.opcodeBinaryString(opcode_value);

        // Getting the register
        String pos = operand[0];

        // If the operand is a register
        if (pos.charAt(0) == 'R') {
            // Convert the register value to fit the binary
            String regVal = sp.registerToBinaryString("#" +
pos.substring(1), line);
            binaryString += regVal;
        }

        // If the operand is an absolute symbol
        else {
            // Ensure the symbol is in the symbol table
            absoluteSymbolTable(pos, line);

            // Convert the register value to fit the binary
            String regVal =
sp.registerToBinaryString(symbolTable.get(pos)[0], line);
            binaryString += regVal;
        }
```

```
        String address = operand[1];

        int relative = 0;

        // name of the external symbol for the modification record
        String externalSym = address;

        // If the given address is a literal

        if (address.charAt(0) == '=') {
            address = sp.literalValue(this.literalTable, address, line);
            relative = 1;
        }

        // If the given address is a symbol
        else if (!(address.charAt(0) == '#' || address.charAt(0) == 'x'))
{
            /*
             * Ensure the symbol is present in the table and check if the
symbol is absolute
             * or relative
             */
            relative = isRelativeSymbol(address, line);

            // Update address if it is a .EXT symbol
            if (relative == 2) {
                address = this.tempExternalTable.get(address);
            } else {
                String val = this.symbolTable.get(address)[0];

                // Check if the address is a value
                if (val.charAt(0) == 'x' || val.charAt(0) == '#') {
                    address = val;
                }

                // otherwise, it is a external symbol
                else {
```

```
                    // make sure the symbol is a external forward
referencing
                    if (!this.tempExternalTable.containsKey(val)) {
                        throw new Pass2Exception("Line: " + line + ":
cannot forward reference a non-external symbol");
                    }

                    // update the value with the value from the external
symbol table
                    address = this.tempExternalTable.get(val);
                    externalSym = val;

                    // when creating modification records, make sure it is
a external symbol
                    relative = 2;
                }
            }
        }

        /*
         * Check if the given address is within page range (with the
location counter)
         * and if so, return the lower 9 bits of the given address
         */
        String offset = sp.pageRangeAndOffset(lcAddress, address, line);
        binaryString += offset;

        // Convert the binary string into a hex string
        String hexString = sp.convertBinaryToHexString(binaryString,
line);
        textRecord += hexString;

        // Check if the symbol was a relative symbol to add a modification
record
        if (this.isRelocatable) {
            if (relative == 1) {
                textRecord += ("X9" + this.segName);
            } else if (relative == 2) {
                textRecord += ("X9" + externalSym);
            }
```

```
        }

        return textRecord;
    }

    /**
     * Forming the text record for the LDI instruction.
     *
     * @param operand      each of the operand split into a separate
string
     * @param opcode_value the size by which LC is to be updated
     * @param size         the size by which LC is to be updated
     * @param line         the current line in the assembly code being
parsed
     * @return the text record for the LDI instruction
     * @throws Pass2Exception throwing a runtime error when a fatal error
is
     *                        detected in Pass2
     */
    private String ldiOp(String[] operand, int opcode_value, int size,
String line) throws Pass2Exception {
        // Calling the instruction with the same procedure
        return ldOp(operand, opcode_value, size, line);
    }

    /**
     * Forming the text record for the LDR instruction.
     *
     * @param operand      each of the operand split into a separate
string
     * @param opcode_value the size by which LC is to be updated
     * @param size         the size by which LC is to be updated
     * @param line         the current line in the assembly code being
parsed
     * @return the text record for the LDR instruction
     * @throws Pass2Exception throwing a runtime error when a fatal error
is
     *                        detected in Pass2
     */
```

```
    private String ldrOp(String[] operand, int opcode_value, int size,
String line) throws Pass2Exception {
        String textRecord = "";

        // Forming the address based on location counter
        String lcAddress = addressFromLC(size);
        textRecord += lcAddress;

        String binaryString = sp.opcodeBinaryString(opcode_value);

        // Iterating through the operand length
        for (int i = 0; i < operand.length; i++) {
            String pos = operand[i];

            // If the operand is a register
            if (pos.charAt(0) == 'R') {
                // Convert the register value to fit the binary
                String regVal = sp.registerToBinaryString("#" +
pos.substring(1), line);
                binaryString += regVal;
            }

            // If the operand is an index
            else if (pos.charAt(0) == '#' || pos.charAt(0) == 'x') {
                String indVal = sp.indexToBinaryString(pos, line);
                binaryString += indVal;
            }

            // If the operand is an absolute symbol
            else {
                // Ensure the symbol is in the symbol table
                absoluteSymbolTable(pos, line);

                // If there is a symbol at the end, it is treated as an
index
                if (i == 2) {
                    String indVal =
sp.indexToBinaryString(symbolTable.get(pos)[0], line);
                    binaryString += indVal;
                }
```

```
                // Otherwise, treat it is a register
                else {
                    // Convert the register value to fit the binary
                    String regVal =
sp.registerToBinaryString(symbolTable.get(pos)[0], line);
                    binaryString += regVal;
                }
            }
        }

        // Convert the binary string into a hex string
        String hexString = sp.convertBinaryToHexString(binaryString,
line);
        textRecord += hexString;

        return textRecord;
    }

    /**
     * Forming the text record for the LEA instruction.
     *
     * @param operand      each of the operand split into a separate
string
     * @param opcode_value the size by which LC is to be updated
     * @param size         the size by which LC is to be updated
     * @param line         the current line in the assembly code being
parsed
     * @return the text record for the LEA instruction
     * @throws Pass2Exception throwing a runtime error when a fatal error
is
     *                        detected in Pass2
     */
    private String leaOp(String[] operand, int opcode_value, int size,
String line) throws Pass2Exception {
        // Calling the instruction with the same procedure
        return ldOp(operand, opcode_value, size, line);
    }

    /**
```

```
     * Forming the text record for the NOT instruction.
     *
     * @param operand      each of the operand split into a separate
string
     * @param opcode_value the size by which LC is to be updated
     * @param size         the size by which LC is to be updated
     * @param line         the current line in the assembly code being
parsed
     * @return the text record for the NOT instruction
     * @throws Pass2Exception throwing a runtime error when a fatal error
is
     *                        detected in Pass2
     */
    private String notOp(String[] operand, int opcode_value, int size,
String line) throws Pass2Exception {
        String textRecord = "";

        // Forming the address based on location counter
        String lcAddress = addressFromLC(size);
        textRecord += lcAddress;

        String binaryString = sp.opcodeBinaryString(opcode_value);

        // Iterating through the operand length
        for (int i = 0; i < operand.length; i++) {
            String pos = operand[i];

            // If the operand is a register
            if (pos.charAt(0) == 'R') {
                // Convert the register value to fit the binary
                String regVal = sp.registerToBinaryString("#" +
pos.substring(1), line);
                binaryString += regVal;
            }

            // If the operand is an absolute symbol
            else {
                // Ensure the symbol is in the symbol table
                absoluteSymbolTable(pos, line);
```

```
                // Convert the register value to fit the binary
                String regVal =
sp.registerToBinaryString(symbolTable.get(pos)[0], line);
                binaryString += regVal;
            }
        }

        // Add the do not care values
        binaryString += "000000";

        // Convert the binary string into a hex string
        String hexString = sp.convertBinaryToHexString(binaryString,
line);
        textRecord += hexString;

        return textRecord;
    }

    /**
     * Forming the text record for the RET instruction.
     *
     * @param opcode_value the size by which LC is to be updated
     * @param size         the size by which LC is to be updated
     * @param line         the current line in the assembly code being
parsed
     * @return the text record for the RET instruction
     */
    private String retOp(int opcode_value, int size, String line) {
        String textRecord = "";

        // Forming the address based on location counter
        String lcAddress = addressFromLC(size);
        textRecord += lcAddress;

        String binaryString = sp.opcodeBinaryString(opcode_value);

        // Adding the don't cares
        binaryString += "000000000000";

        // Convert the binary string into a hex string
```

```
        String hexString = sp.convertBinaryToHexString(binaryString,
line);
        textRecord += hexString;

        return textRecord;
    }

    /**
     * Forming the text record for the LD instruction.
     *
     * @param operand      each of the operand split into a separate
string
     * @param opcode_value the size by which LC is to be updated
     * @param size         the size by which LC is to be updated
     * @param line         the current line in the assembly code being
parsed
     * @return the text record for the LD instruction
     * @throws Pass2Exception throwing a runtime error when a fatal error
is
     *                        detected in Pass2
     */
    private String stOp(String[] operand, int opcode_value, int size,
String line) throws Pass2Exception {
        // Calling the instruction with the same procedure
        return ldOp(operand, opcode_value, size, line);
    }

    /**
     * Forming the text record for the STI instruction.
     *
     * @param operand      each of the operand split into a separate
string
     * @param opcode_value the size by which LC is to be updated
     * @param size         the size by which LC is to be updated
     * @param line         the current line in the assembly code being
parsed
     * @return the text record for the LD instruction
     * @throws Pass2Exception throwing a runtime error when a fatal error
is
     *                        detected in Pass2
```

```
     */
    private String stiOp(String[] operand, int opcode_value, int size,
String line) throws Pass2Exception {
        // Calling the instruction with the same procedure
        return ldOp(operand, opcode_value, size, line);
    }

    /**
     * Forming the text record for the STR instruction.
     *
     * @param operand      each of the operand split into a separate
string
     * @param opcode_value the size by which LC is to be updated
     * @param size         the size by which LC is to be updated
     * @param line         the current line in the assembly code being
parsed
     * @return the text record for the STR instruction
     * @throws Pass2Exception throwing a runtime error when a fatal error
is
     *                        detected in Pass2
     */
    private String strOp(String[] operand, int opcode_value, int size,
String line) throws Pass2Exception {
        // Calling the instruction with the same procedure
        return ldrOp(operand, opcode_value, size, line);
    }

    /**
     * Forming the text record for the TRAP instruction.
     *
     * @param operand      each of the operand split into a separate
string
     * @param opcode_value the size by which LC is to be updated
     * @param size         the size by which LC is to be updated
     * @param line         the current line in the assembly code being
parsed
     * @return the text record for the TRAP instruction
     * @throws Pass2Exception throwing a runtime error when a fatal error
is
     *                        detected in Pass2
```

```
     */
    private String trapOp(String[] operand, int opcode_value, int size,
String line) throws Pass2Exception {
        String textRecord = "";

        // Forming the address based on location counter
        String lcAddress = addressFromLC(size);
        textRecord += lcAddress;

        String binaryString = sp.opcodeBinaryString(opcode_value);

        // Adding the don't cares
        binaryString += "0000";

        // Getting the register
        String pos = operand[0];

        // If the operand is a register
        if (pos.charAt(0) == '#' || pos.charAt(0) == 'x') {
            String trap = sp.trapToBinaryString(pos, line);
            binaryString += trap;
        }

        // If the operand is an absolute symbol
        else {
            // Ensure the symbol is in the symbol table
            absoluteSymbolTable(pos, line);

            // If there is a symbol at the end, it is treated as a trap
vector
            String trap = sp.trapToBinaryString(symbolTable.get(pos)[0],
line);
            binaryString += trap;
        }

        // Convert the binary string into a hex string
        String hexString = sp.convertBinaryToHexString(binaryString,
line);
        textRecord += hexString;
```

```
        return textRecord;
    }

    /**
     * Forming the text record for the .BLKW instruction and outputting to
the
     * listing file.
     *
     * @param symbol  any symbol pertaining to the given line of assembly
     * @param opcode  the name of the instruction
     * @param operand each of the operand split into a separate string
     * @param line    the current line in the assembly code being parsed
     * @return the text record for the LD instruction
     * @throws Pass2Exception throwing a runtime error when a fatal error
is
     *                        detected in Pass2
     */
    private String blkwOp(String symbol, String opcode, String operand,
String line) throws Pass2Exception {
        String textRecord = "";
        int blockVal;

        if (!(operand.charAt(0) == 'x' || operand.charAt(0) == '#')) {
            absoluteSymbolTable(operand, line);
            blockVal = pot.blockLength(this.symbolTable.get(operand)[0]);
        }

        // Otherwise, the value is defined as a constant
        else {
            blockVal = pot.blockLength(operand);
        }

        if (blockVal < 0) {
            throw new Pass2Exception(
                    "Line: " + line + ": The given BLKW value is not
between [1, 65535] or [x1, xFFFF].");
        }

        // Updating the address based on the location counter
        String address = addressFromLC(blockVal);
```

```
        blkwListingFile(symbol, address, opcode, operand, line);

        return textRecord;
    }

    /**
     * Forming the end record for the .END instruction.
     *
     * @param operand each of the operand split into a separate string
     * @param line    the current line in the assembly code being parsed
     * @return the end record for the .END instruction
     * @throws Pass2Exception throwing a runtime error when a fatal error
is
     *                        detected in Pass2
     */
    private String endOp(String operand, String line) throws
Pass2Exception {
        String textRecord = "E";

        // If the address does not end with a value
        if (operand.isEmpty()) {
            textRecord += this.initialAddress;
        } else {
            // If the operand is a hex/decimal value
            if (operand.charAt(0) == 'x' || operand.charAt(0) == '#') {
                textRecord += sp.addressToHex(operand, line);
            }

            // If the operand is a symbol
            else {
                // Verifying the symbol is present in the symbol table
                int relative = isRelativeSymbol(operand, line);
                String address;

                // Update address if it is a .EXT symbol or symbol table
value
                if (relative == 2) {
                    address = this.tempExternalTable.get(operand);
                } else {
                    String val = this.symbolTable.get(operand)[0];
```

```
                    // Check if the address is a value
                    if (val.charAt(0) == 'x' || val.charAt(0) == '#') {
                        address = val;
                    }

                    // otherwise, it is a external symbol
                    else {

                        // make sure the symbol is a external forward
referencing
                        if (!this.tempExternalTable.containsKey(val)) {
                            throw new Pass2Exception(
                                    "Line: " + line + ": cannot forward
reference a non-external symbol");
                        }

                        // update the value with the value from the
external symbol table
                        address = this.tempExternalTable.get(val);
                    }
                }

                textRecord += sp.addressToHex(address, line);
            }
        }

        return textRecord;
    }

    /**
     * Parsing through .ENT operands and forming N records.
     *
     * @param operand each of the operand split into a separate string
     * @param line    the current line in the assembly code being parsed
     * @return the end record for the .END instruction
     * @throws Pass2Exception throwing a runtime error when a fatal error
is
     *                        detected in Pass2
     */
```

```
    private String entOp(String[] operand, String line) throws
Pass2Exception {
        String nRecord = "";

        // Throw an error for > 5 fields in the operand
        if (operand.length > 5) {
            throw new Pass2Exception("Line " + line + ": More than 5 entry
symbols defined in the operand field");
        }

        // Parsing through all the operands
        for (int i = 0; i < operand.length; i++) {
            nRecord += "N";
            String pos = operand[i];

            // Making sure the symbol is relative
            if (isRelativeSymbol(pos, line) != 1) {
                throw new Pass2Exception("Line " + line + ": Symbol \"" +
pos + "\" is not a relative symbol");
            }

            // Getting the symbol value
            String symVal = this.symbolTable.get(pos)[0].substring(1);

            // Adding the symbol name and the values (without hex/decimal)
to the record
            nRecord += (pos + "=" + symVal);
            nRecord += "\n";
        }

        return nRecord;
    }

    /**
     * Parsing .EXT operations and adding any external symbols to a
temporary
     * storage unit.
     *
     * @param operand each of the operand split into a separate string
     * @param line    the current line in the assembly code being parsed
```

```
     * @return the end record for the .END instruction
     * @throws Pass2Exception throwing a runtime error when a fatal error
is
     *                        detected in Pass2
     */
    private String extOp(String[] operand, String line) throws
Pass2Exception {
        String textRecord = "";

        // Throw an error for > 5 fields in the operand
        if (operand.length > 5) {
            throw new Pass2Exception("Line " + line + ": More than 5
external symbols defined in the operand field");
        }

        // Parsing through all the operands
        for (int i = 0; i < operand.length; i++) {
            String pos = operand[i];

            // If it is a symbol used in the program, then throw an error
            if (this.symbolTable.containsKey(pos)) {
                throw new Pass2Exception("Line " + line + ": Symbol \"" +
pos
                        + "\" is already defined in the symbol table
(cannot be used as an external symbol)");
            }

            // Add in the value temporarily with 0
            this.tempExternalTable.put(pos, "x0000");
        }

        return textRecord;
    }

    /**
     * Ensuring the .EQU instruction values are correct.
     *
     * @param operand each of the operand split into a separate string
     * @param line    the current line in the assembly code being parsed
     * @return the text record for the EQU instruction
```

```
     * @throws Pass2Exception throwing a runtime error when a fatal error
is
     *                        detected in Pass2
     */
    private String equOp(String operand, String line) throws
Pass2Exception {
        String textRecord = "";

        if (operand.charAt(0) == 'x' || operand.charAt(0) == '#') {
            if (!sp.canParseInt(operand)) {
                throw new Pass2Exception("Line " + line + ": Invalid
decimal/hex value");

            }
        } else {
            isRelativeSymbol(operand, line);
        }

        return textRecord;
    }

    /**
     * Forming the text record for the .FILL instruction
     *
     * @param operand each of the operand split into a separate string
     * @param line    the current line in the assembly code being parsed
     * @return the text record for the .FILL instruction
     * @throws Pass2Exception throwing a runtime error when a fatal error
is
     *                        detected in Pass2
     */
    private String fillOp(String operand, String line) throws
Pass2Exception {
        String textRecord = "T";
        int relocatable = 0;
        // name of the external symbol for the modification record
        String externalSym = operand;

        // Forming the address based on location counter
        String lcAddress = addressFromLC(1);
```

```
        textRecord += lcAddress;

        // If the operand is a hex/decimal
        if (operand.charAt(0) == 'x' || operand.charAt(0) == '#') {
            textRecord += sp.fillHexString(operand, line);
        }

        // If the operand is a symbol
        else {
            relocatable = isRelativeSymbol(operand, line);
            String address;

            // Update address if it is a .EXT symbol or symbol table value
            if (relocatable == 2) {
                address = this.tempExternalTable.get(operand);
            } else {
                String val = this.symbolTable.get(operand)[0];

                // Check if the address is a value
                if (val.charAt(0) == 'x' || val.charAt(0) == '#') {
                    address = val;
                }

                // otherwise, it is a external symbol
                else {

                    // make sure the symbol is a external forward
referencing
                    if (!this.tempExternalTable.containsKey(val)) {
                        throw new Pass2Exception("Line: " + line + ":
cannot forward reference a non-external symbol");
                    }

                    // update the value with the value from the external
symbol table
                    address = this.tempExternalTable.get(val);
                    externalSym = val;

                    // when creating modification records, make sure it is
a external symbol
```

```
                    relocatable = 2;
                }
            }

            textRecord += sp.fillHexString(address, line);
        }

        // Check if the operand was relocatable
        if (this.isRelocatable) {
            if (relocatable == 1) {
                textRecord += ("X16" + this.segName);
            } else if (relocatable == 2) {
                textRecord += ("X16" + externalSym);
            }
        }

        textRecord += "\n";

        return textRecord;
    }

    /**
     * Forming the header record for the .ORIG instruction.
     *
     * @param symbol  any symbol pertaining to the given line of assembly
     * @param operand each of the operand split into a separate string
     * @param line    the current line in the assembly code being parsed
     * @return the header record for the .ORIG instruction
     * @throws Pass2Exception throwing a runtime error when a fatal error
is
     *                        detected in Pass2
     */
    private String origOp(String symbol, String operand, String line)
throws Pass2Exception {
        String headerRecord = "H";

        // Making the segment name 6 characters long
        while (symbol.length() < 6) {
            symbol += " ";
        }
```

```
        headerRecord += symbol;
        this.segName = symbol;

        // If the origin is loaded at 0 or given
        if (operand.length() < 1) {
            headerRecord += "0000";
            this.isRelocatable = true;
        } else {
            // Making the hex value for the origin
            String originAddress = sp.addressToHex(operand, line);
            this.LC = Integer.parseUnsignedInt(originAddress, 16);

            headerRecord += originAddress;
            this.initialAddress = originAddress;
        }

        // Making sure the segment size is within bounds
        if (this.segmentSize > 0xFFFF) {
            throw new Pass2Exception("Line " + line + ": Segment size is
greater than 0xFFFF");

        }

        // Relocatable and > page length
        if (this.isRelocatable && this.segmentSize > 512) {
            throw new Pass2Exception("Line " + line + ": Segment size is
greater than a page length");
        }

        // Loading the segment size
        String segmentLength = Integer.toUnsignedString(this.segmentSize,
16).toUpperCase();

        // Making the segment size address is 4 characters long
        while (segmentLength.length() < 4) {
            segmentLength = "0" + segmentLength;
        }

        headerRecord += (segmentLength + "\n");
```

```
        return headerRecord;
    }

    /**
     * Forming the text record(s) for the .STRZ instruction and outputting
to the
     * listing file.
     *
     * @param symbol  any symbol pertaining to the given line of assembly
     * @param opcode  the name of the instruction
     * @param operand each of the operand split into a separate string
     * @param line    the current line in the assembly code being parsed
     * @return the text record(s) for the .STRZ instruction
     * @throws Pass2Exception throwing a runtime error when a fatal error
is
     *                        detected in Pass2
     */
    private String strzOp(String symbol, String opcode, String operand,
String line) throws Pass2Exception {
        String finalTextRecord = "";
        String textRecord = "";
        int i = 0;

        for (i = 0; i < operand.length(); i++) {
            // Creating the text record
            textRecord = "T";

            // Forming the address based on location counter
            String lcAddress = addressFromLC(1);
            textRecord += lcAddress;

            // Parsing each character with its ASCII values
            int c = (int) operand.charAt(i);
            String hexString = Integer.toHexString(c).toUpperCase();
            while (hexString.length() < 4) {
                hexString = "0" + hexString;
            }

            // Add the a hex string to the text record
            textRecord += hexString;
```

```
            textRecord += "\n";

            finalTextRecord += textRecord;

            strzListingFile(symbol, opcode, operand, i, textRecord, line);
        }

        textRecord = "T";

        // Forming the address based on location counter
        String lcAddress = addressFromLC(1);
        textRecord += lcAddress;

        // Adding the null termination to the text record
        textRecord += "0000";
        textRecord += "\n";

        finalTextRecord += textRecord;

        strzListingFile(symbol, opcode, operand, i, textRecord, line);

        return finalTextRecord;
    }

    /**
     * Check whether a symbol is absolute or not.
     *
     * @param pos  the symbol to check its relativity
     * @param line the current line in the assembly code being parsed
     * @return the boolean value for whether the symbol is absolute or not
     * @throws Pass2Exception throwing a runtime error when a fatal error
is
     *                        detected in Pass2
     */
    private void absoluteSymbolTable(String pos, String line) throws
Pass2Exception {
        if (this.tempExternalTable.containsKey(pos)) {
            throw new Pass2Exception(
                    "Line " + line + ": External symbol \"" + pos + "\"
cannot be used as an absolute symbol");
```

```
        }

        if (!this.symbolTable.containsKey(pos)) {
            throw new Pass2Exception("Line " + line + ": Symbol \"" + pos
+ "\" is not found in the symbol table");
        }

        if (!symbolTable.get(pos)[1].equals("A")) {
            throw new Pass2Exception("Line " + line + ": Symbol \"" + pos
+ "\" is not an absolute symbol");
        }
    }

    /**
     * Check whether a symbol is relative or not. Returns 1 if the symbol
is in the
     * symbol table and is an external symbol, return 2 if the symbol is
in the
     * external symbol, otherwise it is absolute symbol.
     *
     * @param pos  the symbol to check its relativity
     * @param line the current line in the assembly code being parsed
     * @return the boolean value for whether the symbol is relative or not
     * @throws Pass2Exception throwing a runtime error when a fatal error
is
     *                        detected in Pass2
     */
    private int isRelativeSymbol(String pos, String line) throws
Pass2Exception {
        // Throw an error if it is not contained in either tables
        if (!this.symbolTable.containsKey(pos) &&
!this.tempExternalTable.containsKey(pos)) {
            throw new Pass2Exception("Line " + line + ": Symbol \"" + pos
+ "\" is not found in the symbol table");
        }

        // Symbol table relative
        if (this.tempExternalTable.containsKey(pos)) {
            return 2; // is an external symbol
        } else if (symbolTable.get(pos)[1].equals("R")) {
```

```
            return 1; // is a relative symbol
        }

        return 0; // is an absolute symbol
    }

    /**
     * Returns the address formed by the location counter and increments
it by the
     * size according to the instruction.
     *
     * @param size the size of the instruction
     * @return the address LC points to
     */
    private String addressFromLC(int size) {
        String address = Integer.toHexString(this.LC).toUpperCase();

        while (address.length() < 4) {
            address = "0" + address;
        }

        this.LC = (this.LC + size) % 0x10000;
        return address;
    }

}
```

String_Parser.java

```
package lab3_integrated.assembler.Passes;

import java.util.Map;

import lab3_integrated.assembler.lab2.Exceptions.Pass2Exception;

public class String_Parser {
    /**
```

```
     * Constants defined as finals.
     */
    // Magic numbers
    public static final int ZERO = 0;
    public static final int ONE = 1;
    public static final int TWO = 2;
    public static final int DECIMAL = 10;
    public static final int HEX = 16;
    public static final int INVALID_NUMBER = -0xFFFF;

    // Imm5 sizes
    public static final int IMMEDIATE_HEX_MAX = 0x1F;
    public static final int IMMEDIATE_DEC_MIN = -16;
    public static final int IMMEDIATE_DEC_MAX = 15;

    // Ind6 sizes
    public static final int INDEX_HEX_MAX = 0x3F;
    public static final int INDEX_DEC_MAX = 63;

    // Trapvect8 sizes
    public static final int TRAP_HEX_MAX = 0xFF;
    public static final int TRAP_DEC_MAX = 255;

    // Address and word sizes
    public static final int ADDRESS_HEX_MAX = 0xFFFF;
    public static final int ADDRESS_DEC_MAX = 65535;
    public static final int WORD_MIN = -32768;
    public static final int WORD_MAX = 32767;

    /**
     * Checks whether the given string can be parsed into an integer if it
is within
     * the range defined in the abstract machine.
     *
     * @param input potential string to be parsed into an integer value
     * @return if the given integer can be parsed to an integer if within
the range
     *         of [-32768, 0xFFFF]
     * @ensures canParseInt = Integer.parseInt(input) and within range is
true,
```

```
     *          false otherwise
     */
    public boolean canParseInt(String input) {
        int length = input.length();

        // Return false if the length does not contain a literal/hex
        if (length < TWO) {
            return false;
        }

        char pos0 = input.charAt(ZERO);
        char pos1 = input.charAt(ONE);

        int index = input.length();

        // When the string is a literal
        if (pos0 == '=' && length > TWO) {

            // When the value is hex or decimal
            if (pos1 == 'x') {
                return isInteger(input.substring(TWO, index), HEX);
            } else if (pos1 == '#') {
                return isInteger(input.substring(TWO, index), DECIMAL);
            }
        }

        // When the value is hex or decimal
        else if (pos0 == 'x') {
            return isInteger(input.substring(ONE, index), HEX);
        } else if (pos0 == '#') {
            return isInteger(input.substring(ONE, index), DECIMAL);
        }

        return false;
    }

    /**
     * Attempts to parse and return an immediate value if given in the
correct
     * range.
```

```
     *
     * @param input potential string to be parsed into an integer value
     * @return integer value if within the range of [-16, 15] or [0x0,
0x1F], or
     *         returns invalid number (-0xFFFF)
     * @requires input.length() > 1
     * @ensure parseImmediate = Integer.parseInt(input) within range,
-0xFFFF
     *         otherwise
     */
    public int parseImmediate(String input) {
        int val = parseInt(input);

        if ((isHex(input, ZERO) && val >= ZERO && val <=
IMMEDIATE_HEX_MAX)
                || (isDecimal(input, ZERO) && val >= IMMEDIATE_DEC_MIN &&
val <= IMMEDIATE_DEC_MAX)) {
            return val;
        }

        return INVALID_NUMBER;
    }

    /**
     * Attempts to parse and return an index value if given in the correct
range.
     *
     * @param input potential string to be parsed into an integer value
     * @return integer value if within the range of [0, 63] or [0x0,
0x3F], or
     *         returns invalid number (-0xFFFF)
     * @ensure parseIndex = Integer.parseInt(input) within range, -0xFFFF
otherwise
     */
    public int parseIndex(String input) {
        int val = parseInt(input);

        if ((isHex(input, ZERO) && val >= ZERO && val <= INDEX_HEX_MAX)
                || (isDecimal(input, ZERO) && val >= ZERO && val <=
INDEX_DEC_MAX)) {
```

```
            return val;
        }

        return INVALID_NUMBER;
    }

    /**
     * Attempts to parse and return an trap vector value if given in the
correct
     * range.
     *
     * @param input potential string to be parsed into an integer value
     * @return integer value if within the range of [0, 255] or [0x0,
0xFF], or
     *         returns invalid number (-0xFFFF)
     * @requires input.length() > 1
     * @ensure parseTrapvect = Integer.parseInt(input) within range,
-0xFFFF
     *         otherwise
     */
    public int parseTrapvect(String input) {
        int val = parseInt(input);

        if ((isHex(input, ZERO) && val >= ZERO && val <= TRAP_HEX_MAX)
                || (isDecimal(input, ZERO) && val >= ZERO && val <=
TRAP_DEC_MAX)) {
            return val;
        }

        return INVALID_NUMBER;
    }

    /**
     * Attempts to parse and return an address value if given in the
correct range.
     *
     * @param input potential string to be parsed into an integer value
     * @return integer value if within the range of [0, 65535] or [0x0,
0xFFFF], or
     *         returns invalid number (-0xFFFF)
```

```
     * @requires input.length() > 1
     * @ensure parseAddress = Integer.parseInt(input) within range,
-0xFFFF
     *         otherwise
     */
    public int parseAddress(String input) {
        int val = parseInt(input);

        if ((isHex(input, ZERO) && val >= ZERO && val <= ADDRESS_HEX_MAX)
                || (isDecimal(input, ZERO) && val >= ZERO && val <=
ADDRESS_DEC_MAX)) {
            return val;
        }

        return INVALID_NUMBER;
    }

    /**
     * Checks whether the given string can be parsed into an literal if it
is within
     * the range.
     *
     * @param input potential string to be parsed into a literal value
     * @return if the given integer can be parsed to an integer if within
the range
     *         of [-32768, 32767] or [0x0, 0xFFFF]
     * @ensures isLiteral = Integer.parseInt(input) and within range is
true, false
     *          otherwise
     */
    public boolean isLiteral(String input) {
        boolean b = input.length() > TWO && input.charAt(ZERO) == '=' &&
canParseInt(input);
        if (b) {
            int val = parseInt(input);
            return (isHex(input, ONE) && val >= ZERO && val <=
ADDRESS_HEX_MAX)
                    || (isDecimal(input, ONE) && val >= WORD_MIN && val <=
WORD_MAX);
        }
```

```
        return false;
    }

    /**
     * Attempts to parse and return a literal value as a string if given
in the
     * correct range.
     *
     * @param input potential string to be parsed into a literal value
     * @return if the given integer can be parsed to an integer if within
the range
     *         of [-32768, 32767] or [0x0, 0xFFFF], -0xFFFF otherwise
     * @ensures isLiteral = Integer.parseInt(input) and within range, null
otherwise
     */
    public String parseLiteralString(String input) {
        int val = parseInt(input);

        if ((isHex(input, ONE) && val >= ZERO && val <= ADDRESS_HEX_MAX)
                || (isDecimal(input, ONE) && val >= WORD_MIN && val <=
WORD_MAX)) {
            String s = Integer.toHexString(val);

            if (s.length() <= 4) {
                while (s.length() < 4) {
                    s = "0" + s;
                }
            } else {
                s = s.substring(s.length() - 4, s.length());
            }

            return s;
        }

        return null;
    }

    /**
     * Attempts to parse and return an union range value if given in the
correct
```

```
     * range.
     *
     * @param input potential string to be parsed into an integer value
     * @return integer value if within the range of [-32678, 32677] or
[0x0,
     *         0xFFFF], or returns invalid number (-0xFFFF)
     * @requires input.length() > 1
     * @ensure parseUnionRange = Integer.parseInt(input) within range,
-0xFFFF
     *         otherwise
     */
    public int parseUnionRange(String input) {
        int val = parseInt(input);

        if ((isHex(input, ZERO) && val >= ZERO && val <= ADDRESS_HEX_MAX)
                || (isDecimal(input, ZERO) && val >= WORD_MIN && val <=
WORD_MAX)) {
            return val;
        }

        return INVALID_NUMBER;
    }

    /**
     * Returns the binary string version of the index6 value.
     *
     * @param ind  the index value to be parsed
     * @param line the current line of assembly code
     * @return the index value as a binary string
     * @throws Pass2Exception
     */
    public String indexToBinaryString(String ind, String line) throws
Pass2Exception {
        if (!canParseInt(ind)) {
            throw new Pass2Exception("Line " + line + ": Invalid index6
value");
        }

        int index = parseIndex(ind);
```

```
        if (index < -0x8000) {
            throw new Pass2Exception(
                    "Line " + line + ": Index6 value is not within the
specified range [#0 - #63] or [0x0 - 0x3F]");
        }

        String s = Integer.toBinaryString(index);
        while (s.length() < 6) {
            s = "0" + s;
        }

        return s;
    }

    /**
     * Returns if a given address is within the page range as the location
counter.
     *
     * @param lc      the location counter
     * @param address the address
     * @param line    the current line of assembly code
     * @return Parses the given address and ensures the address is within
the same
     *         page range as location counter
     * @throws Pass2Exception
     */
    public String pageRangeAndOffset(String lc, String address, String
line) throws Pass2Exception {
        if (!canParseInt(address)) {
            throw new Pass2Exception("Line " + line + ": Invalid address
value");
        }

        int add_val = parseAddress(address);

        if (add_val < -0x8000) {
            throw new Pass2Exception("Line " + line
                    + ": Address value is not within the specified range
[#0 - #65535] or [0x0 - 0xFFFF]");
        }
```

```
        // Interpret the LC address as a hex
        int lc_val = (parseAddress("x" + lc) + 1) % 0x10000;

        int page_lc = (lc_val / 512) % 128;
        int page_address = (add_val / 512) % 128;

        if (page_lc != page_address) {
            throw new Pass2Exception(
                    "Line " + line + ": Address value is not within the
same page number as PC (PC at page: #" + page_lc
                            + ", Defined Address at page: #" +
page_address + ")");

        }

        String s = Integer.toBinaryString(add_val);
        while (s.length() < 16) {
            s = "0" + s;
        }

        return s.substring(7, s.length());
    }

    /**
     * Returns the opcode's value as a binary string.
     *
     * @param opcode_value the decimal value for the machine instructions
     * @return the binary string of the opcode value
     */
    public String opcodeBinaryString(int opcode_value) {
        String value = Integer.toBinaryString(opcode_value);

        while (value.length() < 4) {
            value = "0" + value;
        }

        return value;
    }
```

```
    /**
     * Returns the binary string version of the register value.
     *
     * @param reg  the register value to be parsed
     * @param line the current line of assembly code
     * @return the register value as a binary string
     * @throws Pass2Exception
     */
    public String registerToBinaryString(String reg, String line) throws
Pass2Exception {
        // Treat the register value as a assembler decimal
        if (!canParseInt(reg)) {
            throw new Pass2Exception("Line " + line + ": Invalid register
value");
        }

        int reg_val = parseIndex(reg);
        if (reg_val < 0 || reg_val > 7) {
            throw new Pass2Exception("Line " + line + ": Register value
not within range (decimal and hex: [0 - 7])");
        }

        String value = Integer.toBinaryString(reg_val);

        while (value.length() < 3) {
            value = "0" + value;
        }

        return value;
    }

    /**
     * Parses the binary string into a 4 characters hex string.
     *
     * @param binString the binary string
     * @param line      the current line of assembly code
     * @return the hex version of the entire, 16-bit, binary string
     */
    public String convertBinaryToHexString(String binString, String line)
{
```

```
        int bin_val = Integer.parseUnsignedInt(binString, 2);
        String hexValue = Integer.toHexString(bin_val).toUpperCase();

        while (hexValue.length() < 4) {
            hexValue = "0" + hexValue;
        }

        return hexValue;
    }

    /**
     * Returns the binary string version of the immediate value.
     *
     * @param imm  the immediate value to be parsed
     * @param line the current line of assembly code
     * @return the immediate value as a binary string
     * @throws Pass2Exception
     */
    public String immediateToBinaryString(String imm, String line) throws
Pass2Exception {
        if (!canParseInt(imm)) {
            throw new Pass2Exception("Line " + line + ": Invalid immediate
value");
        }

        int immediate = parseImmediate(imm);

        if (immediate < -0x8000) {
            throw new Pass2Exception(
                    "Line " + line + ": Imm5 value is not within the
specified range [#-16 - #15] or [0x0 - 0x1F]");
        }

        String s = Integer.toBinaryString(immediate);
        if (s.length() < 5) {
            while (s.length() < 5) {
                s = "0" + s;
            }
        } else {
            s = s.substring(s.length() - 5, s.length());
```

```
        }

        return s;
    }

    /**
     * Returns the binary string version of the trapvect8 value.
     *
     * @param trap the index value to be parsed
     * @param line the current line of assembly code
     * @return the trap vector value as a binary string
     * @throws Pass2Exception
     */
    public String trapToBinaryString(String trap, String line) throws
Pass2Exception {
        if (!canParseInt(trap)) {
            throw new Pass2Exception("Line " + line + ": Invalid trapvect8
value");
        }

        int trap_vect = parseTrapvect(trap);

        if (trap_vect < -0x8000) {
            throw new Pass2Exception(
                    "Line " + line + ": Trapvect8 value is not within the
specified range [#0 - #255] or [0x0 - 0xFF]");
        }

        String s = Integer.toBinaryString(trap_vect);
        while (s.length() < 8) {
            s = "0" + s;
        }

        return s;
    }

    /**
     * Returns the value of the literal as a hex string.
     *
```

```
     * @param literalTable table containing information about the literal
table
     * @param address      the given address to compare (operand)
     * @param line         the current line of assembly code
     * @return the literal value as a hex string
     * @throws Pass2Exception
     */
    public String literalValue(Map<String, Integer> literalTable, String
address, String line) throws Pass2Exception {
        if (!literalTable.containsKey(address)) {
            throw new Pass2Exception(
                    "Line " + line + ": The literal " + address + " is not
found in the literal table");
        }

        if (!isLiteral(address)) {
            throw new Pass2Exception("Line " + line
                    + ": The given literal value is not within range
[#-32768 - #32767] or [0x0 - 0xFFFF]");
        }

        int litVal = literalTable.get(address);
        String litHex = Integer.toHexString(litVal);

        while (litHex.length() < 4) {
            litHex = "0" + litHex;
        }

        litHex = litHex.toUpperCase();
        litHex = "x" + litHex;
        return litHex;
    }

    /**
     * Returns the hex version of the address in the String.
     *
     * @param operand the operand containing the address
     * @param line    the current line of assembly code
     * @return the given address as a hex value
     * @throws Pass2Exception
```

```
     */
    public String addressToHex(String operand, String line) throws
Pass2Exception {
        if (!canParseInt(operand)) {
            throw new Pass2Exception("Line " + line + ": Invalid address
value");
        }

        int address = parseAddress(operand);

        if (address < -0x8000) {
            throw new Pass2Exception("Line " + line
                    + ": Address value is not within the specified range
[#0 - #65535] or [0x0 - 0xFFFF]");
        }

        String s = Integer.toHexString(address).toUpperCase();
        while (s.length() < 4) {
            s = "0" + s;
        }

        return s;
    }

    /**
     * Returns the hex version of the address in the String (specific for
.FILL).
     *
     * @param operand the operand containing the address
     * @param line    the current line of assembly code
     * @return the given address as a hex value (specific for .FILL)
     * @throws Pass2Exception
     */
    public String fillHexString(String operand, String line) throws
Pass2Exception {
        if (!canParseInt(operand)) {
            throw new Pass2Exception("Line " + line + ": Invalid
decimal/hex value for .FILL");
        }
```

```
        int address = parseUnionRange(operand);

        if (address < -0x8000) {
            throw new Pass2Exception("Line " + line + ": .FILL value not
within range");
        }

        String s = Integer.toHexString(address);

        if (s.length() <= 4) {
            while (s.length() < 4) {
                s = "0" + s;
            }
        } else {
            s = s.substring(s.length() - 4, s.length());
        }

        return s.toUpperCase();
    }

    /**
     * Returns the integer value if a given String can be parsed into an
integer.
     *
     * @param input potential string to be parsed into an integer value
     * @return the parsed integer if within a specified range, -0xFFFF if
not
     *         possible
     * @requires input.length() > 1
     * @ensure parseInt(input) = Integer.parseInt(input) if within range,
-0xFFFF
     *         otherwise
     */
    private int parseInt(String input) {
        int length = input.length();
        int index = input.length();

        char pos0 = input.charAt(ZERO);
        char pos1 = input.charAt(ONE);
```

```
        // When the String is a literal
        if (pos0 == '=' && length > TWO) {

            // When the value is hex or decimal
            if (pos1 == 'x') {
                return Integer.parseInt(input.substring(TWO, index), HEX);
            } else if (pos1 == '#') {
                return Integer.parseInt(input.substring(TWO, index),
DECIMAL);
            }
        }

        // When the value is hex or decimal
        else if (pos0 == 'x') {
            return Integer.parseInt(input.substring(ONE, index), HEX);
        } else if (pos0 == '#') {
            return Integer.parseInt(input.substring(ONE, index), DECIMAL);
        }

        return INVALID_NUMBER;
    }

    /**
     * Checks whether the integer can be parsed into a string, with the
given
     * base-representation is within range.
     *
     * @param input the string containing the number to be parsed into an
integer
     * @param radix the base representation of the integer
     * @return whether the string is in the range defined for radix ([0x0,
0xFFFF]
     *         for base-16 and [-32768, 65535] for base-10)
     * @ensures isInteger = Integer.parseInt(input) within range is true,
false
     *          otherwise
     */
    private boolean isInteger(String input, int radix) {
        try {
            int val = Integer.parseInt(input, radix);
```

```
            // Hex must be positive in the abstract state machine
            if (radix == HEX) {
                return val >= ZERO && val <= ADDRESS_HEX_MAX;
            }

            return val >= WORD_MIN && val <= ADDRESS_DEC_MAX;
        } catch (NumberFormatException e) {
            return false;
        }
    }

    /**
     * Checks whether the given input is a hex number.
     *
     * @param input the potential string that is a hex value
     * @param index the index position at which the character is denoted
as a hex
     *              ('x' character)
     * @return if the string is a hex value or not
     * @ensures isDecimal = input.charAt(index) is 'x' is true, false
otherwise
     */
    private boolean isHex(String input, int index) {
        return input.length() > index && input.charAt(index) == 'x';
    }

    /**
     * Checks whether the given input is a decimal number.
     *
     * @param input the potential string that is a decimal value
     * @param index the index position at which the character is denoted
as a
     *              decimal ('#' character)
     * @return if the string is a decimal value or not
     * @ensures isDecimal = input.charAt(index) is '#' is true, false
otherwise
     */
    private boolean isDecimal(String input, int index) {
        return input.length() > index && input.charAt(index) == '#';
```

```
    }
}
```

Lab2 Package

Exceptions.java

```
package lab3_integrated.assembler.lab2;

public class Exceptions {
    public static class TooShortException extends Exception {
        @Override
        public String getMessage() {
            return "Line too short.";
        }
    }

    public static class CommentLineNoSemicolon extends Exception {
    }

    public static class InvalidOperandException extends Exception {
    }

    public static class Pass2Exception extends Exception {
        /**
         * Default constructor of the exception class.
         *
         * @param message message to print to the console
         */
        public Pass2Exception(String message) {
            super(message);
        }
    }
}
```

Validate.java

```
package lab3_integrated.assembler.lab2;

import java.util.ArrayList;
import java.util.List;

import lab3_integrated.assembler.lab2.Exceptions.CommentLineNoSemicolon;
import lab3_integrated.assembler.lab2.Exceptions.InvalidOperandException;
import lab3_integrated.assembler.MOT.Machine_Op_Table;

public class Validate {
    Machine_Op_Table mot;

    Validate(Machine_Op_Table mot) {
        this.mot = mot;
    }

    enum OpType {
        COMMENT, ORIG, END, EQU, FILL, STRZ, BLKW, INSTRUCTION, UNKNOWN,
ENT, EXT
    }

    enum OperandType {
        REGISTER, IMMEDIATE, INDEX, ADDRESS, NONE, PSEUDOOP, LITERAL,
ENT_EXT
    }

    public static class Locations {
        public static final int LABEL = 0;
        public static final int OPERATION = 1;
        public static final int OPERANDS = 2;
        public static final int LINE = 3;
    }

    private OpType psudeoOP;

    private List<String> fileContainsOrigAndEnd = new ArrayList<>();;
    /**
     * Line number.
```

```
     */

    private int commentOrEntExtCounter = 1;
    /**
     * Validates the given line of code and returns a list that represents
the line
     * containing the a symbol (if applicable), operation, operand, and
line number
     * @param line - The line to validate
     * @param lines - List of all lines in the program
     * @return - A list representing the line in the following order
     * {Symbol, Operation, Operand, Line Number}
     * @throws Exception - If an error occurs during validation
     */
    List<String> validate(String line, ArrayList<List<String>> lines, int
lineCount) throws Exception {

        if(line.isBlank()) {
            throw new IllegalArgumentException("An Empty Line is NOT
valid");
        }

        OpType type = getLineType(line);
        this.psudeoOP = type;

        if (type == OpType.COMMENT) {
            this.commentOrEntExtCounter++;
            return null;
        } else if (type == OpType.UNKNOWN) {
            if (!containsAnyKey(line)) {
                throw new IllegalArgumentException("LINE MUST HAVE PSEUDO
OP OR MACHINE OP");
            }
            validateInstruction(line);
        }

        line = stripComment(line);
        List<String> result = validateAndSplitLine(line, lines);

        if (type == OpType.ENT || type == OpType.EXT) {
```

```
            // validate line content
            // make sure line above it is .ORIG or .EXT or .ENT
            if(lineCount - this.commentOrEntExtCounter != 2) {
                throw new IllegalArgumentException("ENT and EXT must be
declared right after ORIG");
            }
            validateEntOrExt(line,result);
            this.commentOrEntExtCounter++;
        }else if(type != OpType.INSTRUCTION && type != OpType.UNKNOWN){
            boolean isValidOperand =
checkOperands(result.get(Locations.OPERANDS), new OperandType[]
{OperandType.PSEUDOOP});
            if(!isValidOperand) {
                throw new IllegalArgumentException("Invalid Operand");
            }

        }

        if (result.get(Locations.OPERANDS).contains("\"")) {
            String tempString = result.remove(Locations.OPERANDS);
            tempString = tempString.substring(1, tempString.length() - 1);
            result.add(Locations.OPERANDS, tempString);
        }

        result.add(Integer.toString(lineCount));
        validateOperation(result.get(Locations.OPERATION), type);


        if(type == OpType.END || type == OpType.ORIG){
            if(lineCount - this.commentOrEntExtCounter != 1 && type ==
OpType.ORIG) {
                System.out.println(lineCount -
this.commentOrEntExtCounter);
                throw new IllegalArgumentException("ORIG must be the first
pseudo op in the file ");
            }
            fileContainsOrigAndEnd.add(type.toString());
        }

        if(type == OpType.EQU || type == OpType.ORIG){
```

```
            try {
                checkSymbol(result.get(Locations.LABEL).trim());
            }catch(Exception e) {
                throw new IllegalArgumentException("EQU and ORIG must have
a label ");
            }

        }
        return result;
    }

    /**
     * Removes the semicolon from a comment
     * @param line - the comment
     * @return - updated comment without semicolon
     */
    String stripComment(String line) {
        return line.split(";")[0];
    }

    /**
     * Checks if the Input line contains any Machine OP Table Symbol
     * @param input - line to validate
     * @return true if Input contains Machine OP Table Symbol
     */
    boolean containsAnyKey(String input) {
        input = input.trim();
        String[] inputArr = input.split(" ");

        for (String s : inputArr) {
            s = s.trim();
            if (mot.containsOp(s)) {
                return true;
            }
        }

        return false;
    }

    boolean isDebugOrRet(String input) {
```

```
        input = input.trim();
        String[] inputArr = input.split(" ");

        for (String s : inputArr) {
            s = s.trim();
            if (mot.containsOp(s) && (s.equals("DBUG") ||
s.equals("RET"))) {
                return true;
            }
        }

        return false;
    }

    /**
     * Checks if a line is a comment
     * @param line - Comment line
     * @throws Exceptions.CommentLineNoSemicolon if line is not a comment
     */
    void validateComment(String line) throws
Exceptions.CommentLineNoSemicolon {
        if (line.length() > 0 && line.trim().charAt(0) != ';') {
            throw new Exceptions.CommentLineNoSemicolon();
        }
    }

    /**
     * Check is an operation is valid meaning it's a Pseudo OP or Machine
OP
     * @param operation - operation to validate
     * @param type - ENUM of all possible Pseudo OPs
     * @throws Exception if not a valid operation
     */
    void validateOperation(String operation, OpType type) throws Exception
{
        operation = operation.trim();
        if (!operation.equals("." + type) && !mot.containsOp(operation)) {
            throw new IllegalArgumentException("WAS EXPECTING " + "." +
type + " GOT " + operation);
        }
```

```
    }

    /**
     * Determine if a line containing a Machine OP is syntactically valid
     * so Symbol, Operation, Operand and Comment where symbol and comment
are optional
     * @param line - line to syntactically validate
     * @throws InvalidOperandException if validateOperand method fails
     */
    void validateInstruction(String line) throws InvalidOperandException {
        line = line.trim();
        String[] lineArr = line.split("\s+");
        List<String> arr = stripArray(lineArr);
        if (arr.size() == 3) {
            String sym = arr.remove(0);
            checkSymbol(sym);

            String operation = arr.remove(0).trim();
            String operand = arr.remove(0).trim();
            validateOperand(operand, operation);
        } else if (arr.size() == 2) {
            String operation = arr.remove(0).trim();
            String operand = arr.remove(0).trim();
            validateOperand(operand, operation);
        } else {
            if(!isDebugOrRet(lineArr[0])) {
                throw new IllegalArgumentException("The following line is
invalid: " + line);
            }

        }
    }

    /**
     * Removes comment from line if there is one and returns a list of all
other data
     * @param arr - String array containing data from line that's split by
spaces
     * @return - ArrayList containing the remaining data without comment
```

```
     */
    List<String> stripArray(String[] arr) {
        List<String> result = new ArrayList<>();
        for (String s : arr) {
            s = s.trim();
            if (s.length() > 0 && s.charAt(0) == ';') {
                break;
            }
            result.add(s);
        }
        return result;
    }

    /**
     * Determines if operation is a Machine OP then verifies it's operand
length and order
     * is as expected according machine's description
     *
http://web.cse.ohio-state.edu/~giles.25/3903/assignments/project2.html
     * @param operand - operand of the operation
     * @param operation - operation to perform
     * @throws Exceptions.InvalidOperandException if operation and/or
operand are incompatible
     */
    void validateOperand(String operand, String operation) throws
Exceptions.InvalidOperandException {
        operand = operand.trim();
        if (operand.equals("") && !(operation.equals(".ORIG") ||
operation.equals(".END"))) {
            throw new Exceptions.InvalidOperandException();
        }

        switch (operation) {
        case "ADD":
        case "AND":
            if (!checkOperands(operand,
                    new OperandType[] { OperandType.REGISTER,
OperandType.REGISTER, OperandType.REGISTER })
                    && !checkOperands(operand,
```

```
                        new OperandType[] { OperandType.REGISTER,
OperandType.REGISTER, OperandType.IMMEDIATE })) {
                throw new IllegalArgumentException("Invalid Operand " +
operand);
            }
            break;
        case "BR":
        case "BRN":
        case "BRZ":
        case "BRP":
        case "BRNZ":
        case "BRNP":
        case "BRZP":
        case "BRNZP":
        case "JSR":
        case "JMP":
            if (!checkOperands(operand, new OperandType[] {
OperandType.ADDRESS })) {
                throw new IllegalArgumentException("Invalid Operand " +
operand);
            }
            break;
        case "DBUG":
        case "RET":
            if (!checkOperands(operand, new OperandType[] {
OperandType.NONE })) {
                throw new IllegalArgumentException("Invalid Operand " +
operand);
            }
            break;

        case "JSRR":
        case "JMPR":
            if (!checkOperands(operand, new OperandType[] {
OperandType.REGISTER, OperandType.IMMEDIATE })) {
                throw new IllegalArgumentException("Invalid Operand " +
operand);
            }
            break;
        case "LDI":
```

```
        case "LEA":
        case "ST":
        case "STI":
            if (!checkOperands(operand, new OperandType[] {
OperandType.REGISTER, OperandType.ADDRESS })) {
                throw new IllegalArgumentException("Invalid Operand " +
operand);
            }
            break;
        case "LD":
            if (!checkOperands(operand, new OperandType[] {
OperandType.REGISTER, OperandType.ADDRESS })
                    && !checkOperands(operand, new OperandType[] {
OperandType.REGISTER, OperandType.LITERAL })) {
                throw new IllegalArgumentException("Invalid Operand " +
operand);
            }
            break;
        case "LDR":
        case "STR":
            if (!checkOperands(operand,
                    new OperandType[] { OperandType.REGISTER,
OperandType.REGISTER, OperandType.INDEX })) {
                throw new IllegalArgumentException("Invalid Operand " +
operand);
            }
            break;
        case "NOT":
            if (!checkOperands(operand, new OperandType[] {
OperandType.REGISTER, OperandType.REGISTER })) {
                throw new IllegalArgumentException("Invalid Operand " +
operand);
            }
            break;
        case "TRAP":
            if (!checkOperands(operand, new OperandType[] {
OperandType.IMMEDIATE })) {
                throw new IllegalArgumentException("Invalid Operand " +
operand);
            }
```

```
            // DO Something
            break;
        default:

            throw new IllegalArgumentException("Invalid Operation  " +
operation);

        }

    }
    /**
     * Determine if the operand of the Machine OP or Pseudo OP is
syntactically correct
     *
http://web.cse.ohio-state.edu/~giles.25/3903/assignments/project2.html
     * @param operand - to verify
     * @param desiredOps - Array of ENUMS which determine what the
ordering of an operand should be
     * @return - true if the operand is valid
     */
    boolean checkOperands(String operand, OperandType[] desiredOps) {
        String[] op = operand.split(",");
        if (op.length != desiredOps.length && desiredOps[0] !=
OperandType.NONE ) {
            return false;
        }

        for (int i = 0; i < desiredOps.length; i++) {
            op[i] = op[i].trim();
            String currentString = op[i];
            char firstCh = currentString.length() > 0 ?
currentString.charAt(0): ' ';
            OperandType currentType = desiredOps[i];

            if (desiredOps[i] == OperandType.REGISTER) {
                if (firstCh == 'R') {
                    if (currentString.length() > 2 ||
!Character.isDigit(currentString.charAt(1))
                            || currentString.charAt(1) - '0' > 7) {
                        return false;
```

```
                    }
                } else {
                    try {
                    checkSymbol(currentString);

                    }catch(Exception e) {
                        return false;
                    }

                }
            } else if (currentType == OperandType.IMMEDIATE || currentType
== OperandType.INDEX
                    || currentType == OperandType.ADDRESS) {
                if (firstCh == 'x' || firstCh == '#') {
                    String digits = currentString.substring(1);

                    if (!digits.matches("^-?\\d+.*")) {
                        return false;
                    }
                } else if (firstCh == 'R') {
                    return false;
                } else {
                    try {
                        checkSymbol(currentString);
                    }catch(Exception e) {
                            return false;
                        }
                }

            } else if (currentType == OperandType.NONE) {
                if (!currentString.isBlank()) {
                    try{
                           checkSymbol(currentString);
                       }catch(Exception e) {
                           return false;
                       }
                }

            } else if (currentType == OperandType.PSEUDOOP){
//              operand = stripComment(operand);
```

```
                if(this.psudeoOP == OpType.EQU || this.psudeoOP ==
OpType.BLKW) {

                    if (!operand.matches("^x[0-9a-fA-F]+$") &&
!operand.matches("^#-?[0-9]+$")) {
                        try{
                            checkSymbol(operand);
                        }catch(Exception e) {
                            return false;
                        }

                    }

                }else {
                    boolean alphaNumericCheck =
operand.matches("^x[0-9a-fA-F]+$") || operand.matches("^#-?[0-9]+$");
                    if (this.psudeoOP == OpType.ORIG) {
                        if (!alphaNumericCheck && !operand.isBlank() &&
!isComment(operand)){
                            return false;
                        }
                    }else if(this.psudeoOP == OpType.END) {
                        if (!alphaNumericCheck && !operand.isBlank() &&
!isComment(operand)) {
                                try{
                                    checkSymbol(operand);
                                }catch(Exception e) {
                                    return false;
                                }

                            }

                    }else if(this.psudeoOP == OpType.FILL) {
                        if (!alphaNumericCheck) {
                                try{
                                    checkSymbol(operand);
                                }catch(Exception e) {
                                    return false;
                                }
```

```
                            }
                    }
                }


            }else if(currentType == OperandType.LITERAL) {
                if(currentString.length() >= 3 && currentString.charAt(0)
== '=' ) {
                    String literal = currentString.substring(1);
                    if(!literal.matches("^x[0-9a-fA-F]+$") &&
!literal.matches("^#-?[0-9]+$")){
                        return false;
                    }
                }else {
                    return false;
                }


            }else {
                return false;
            }
        }

        return true;
    }

    /**
     * Checks if a symbol already exists
     * @param symbol - to determine if already present
     * @param lines - of all data parsed where list index 0 is the symbol
     */
    void checkIfSymExists(String symbol, ArrayList<List<String>> lines) {
        for (int i = 0; i < lines.size(); i++) {
            if (lines.get(i).get(0).equals(symbol)) {
                throw new IllegalArgumentException("Symbol already
exists");
            }
        }
    }
```

```
    boolean isComment(String line) {
        line = line.trim();
        if(line.length() == 0 || line.charAt(0) != ';') {
            return false;
        }
        return true;
    }

    /**
     * Checks if a string is a valid symbol based on the machine
description
     * @param symbol - to verify
     */
    void checkSymbol(String symbol) {
        if (symbol.length() > 6 || symbol.charAt(0) == 'x') {
            throw new IllegalArgumentException("Error symbol");
        }

        if(symbol.length() > 0 &&
!Character.isAlphabetic(symbol.charAt(0))) {
            throw new IllegalArgumentException("Error symbol");
        }

        if (!symbol.matches("[A-Za-z0-9]+")) {
            throw new IllegalArgumentException("Error symbol");
        }
    }

    /**
     * Splits line and verify's it has all five parts based in the
following order
     * Lable, White Space, Operation, White Space, Operand/Comment
     * @param line - line to validate
     * @param lines - List of all lines in the program
     * @return - A list representing the line in the following order
{Symbol, Operation, Operand}
     * @throws Exception if line doesn't meet requirements
     */
    List<String> validateAndSplitLine(String line, ArrayList<List<String>>
lines) throws Exception {
```

```
        if (line.length() < 17) {
            throw new Exceptions.TooShortException();
        }

        String label = line.substring(0, 6);
        String firstWhiteSpace = line.substring(6, 9);
        String operation = line.substring(9, 14);
        String secondWhiteSpace = line.substring(14, 17);
        String operandsOrComment = line.substring(17);

        label = label.trim();
        operation = operation.trim();
        operandsOrComment = operandsOrComment.trim();

        if (label.length() > 0) {
            checkIfSymExists(label, lines);
            char labelFirstChar = label.charAt(0);
            if (!Character.isAlphabetic(labelFirstChar) || labelFirstChar
== 'R' || labelFirstChar == 'x') {
                throw new IllegalArgumentException(
                        "Label must start with an alphabetic character
that is NOT a R OR an x");
            }
        }

        if (!firstWhiteSpace.isBlank()) {
            throw new IllegalArgumentException("White space must NOT
contain any characters");
        }

        if (!secondWhiteSpace.isBlank()) {
            throw new IllegalArgumentException("White space must NOT
contain any characters");
        }
        List<String> result = new ArrayList<>();

        // operation -> 0, operandsOrComments -> 1 originally
        result.add(Locations.LABEL, label);
        result.add(Locations.OPERATION, operation);
        result.add(Locations.OPERANDS, operandsOrComment);
```

```
        return result;
    }

    /**
     * Returns ENUM describing the type of line passed
     * @param line - to determine type of
     * @return - ENUM of pseudo OP or unknown type
     */
    OpType getLineType(String line) {
        line = line.trim();

        if(line.length() < 0) {
            return OpType.UNKNOWN;
        }

        if (line.charAt(0) == ';') {
            return OpType.COMMENT;
        } else if (line.contains(OpType.STRZ.toString())) {
            return OpType.STRZ;
        } else if (line.contains(OpType.BLKW.toString())) {
            return OpType.BLKW;
        } else if (line.contains(OpType.ORIG.toString())) {
            return OpType.ORIG;
        } else if (line.contains(OpType.EQU.toString())) {
            return OpType.EQU;
        } else if (line.contains(OpType.END.toString())) {
            return OpType.END;
        } else if (line.contains(OpType.FILL.toString())) {
            return OpType.FILL;
        } else if (line.contains(OpType.ENT.toString())) {
            return OpType.ENT;
        } else if (line.contains(OpType.EXT.toString())) {
            return OpType.EXT;
        } else {
            return OpType.UNKNOWN;
        }
    }

    /**
```

```
     * Determine if ORIG and END exist excatly once
     * @return true if ORIG and END both appear once false otherwise
     */
    public boolean doesFileContainsOrigAndEnd() {
        return fileContainsOrigAndEnd.size() == 2? true: false;
    }

    void validateEntOrExt(String line, List<String> list) throws
CommentLineNoSemicolon {
        line = line.trim();


        if(!list.get(0).isBlank()) {
            throw new IllegalArgumentException("No Label Allowed for ENT
or EXT operations");
        }

        if(list.get(1).trim().charAt(0) != ';') {

            String[] symbols = list.get(2).split(",");

            for(String s: symbols) {
                checkSymbol(s.trim());
            }

        }
    }


}
```